Gauss-Jordan

By Jose Divasón

August 2, 2013

Contents

theory *Numeral-Type-Addenda*
imports
$ISABELLE-HOME/src/HOL/Library/Enum
$ISABELLE-HOME/src/HOL/Library/Numeral-Type
$ISABELLE-HOME/src/HOL/Multivariate-Analysis/Cartesian-Euclidean-Space
begin

Code equations for *1*:

lemma [*code abstype*]: *Abs-num1* (*Rep-num1 v*) = (*v*::*num1*) **by** (*metis Rep-num1-inverse*)

lemma [*code abstract*]: *Rep-num1* (*1*) = () **by** *simp*

Code equations for *'a bit0*:

lemma [*code abstype*]: *Abs-bit0* (*Rep-bit0 a*) = *a* **using** *bit0.Rep-inverse* **.**

lemma [*code abstract*]: *Rep-bit0 0* = *0* **unfolding** *bit0.Rep-0* **..**

lemma [*code abstract*]: *Rep-bit0 1* = *1* **unfolding** *bit0.Rep-1* **..**

If the underlying finite type has a computable cardinal, then the representation of elements can be "computed" as follows:

lemma [*code abstract*]:
Rep-bit0 (*Abs-bit0' x* :: *'a* :: {*finite*, *card-UNIV*} *bit0*) = *x mod int* (*CARD*(*'a bit0*))
apply(*simp add*: *Abs-bit0'-def*)
apply(*rule Abs-bit0-inverse*)
apply *simp*
by (*metis bit0.Rep-Abs-mod bit0.Rep-less-n card-bit0 of-nat-numeral zmult-int*)

Code equations for *'a bit1*:

lemma [*code abstype*]: *Abs-bit1* (*Rep-bit1 a*) = *a* **using** *bit1.Rep-inverse* **.**

lemma [*code abstract*]: *Rep-bit1 0* = *0* **unfolding** *bit1.Rep-0* **..**

lemma [*code abstract*]: *Rep-bit1 1* = *1* **unfolding** *bit1.Rep-1* **..**

If the underlying finite type has a computable cardinal, then the representation of elements can be "computed" as follows:

lemma [*code abstract*]:
Rep-bit1 (*Abs-bit1' x* :: *'a* :: {*finite*, *card-UNIV*} *bit1*) = *x mod int* (*CARD*(*'a bit1*))
apply(*simp add*: *Abs-bit1'-def*)
apply(*rule Abs-bit1-inverse*)
apply *simp*
by (*metis of-nat-0-less-iff of-nat-Suc of-nat-mult of-nat-numeral pos-mod-conj zero-less-Suc*)

The *1* is trivially an instance of the equal type class:

instantiation *num1* :: *equal*
begin

definition *equal-num1* :: *num1* => *num1* => *bool*
where *equal-num1 x y* = *True*

instance by (*intro-classes*, *simp add*: *equal-num1-def num1-eq-iff*)
end

The numeral types $'a\ bit0$ and $'a\ bit1$ are instances of the type class equal; would it be better to use *Rep-bit0* in the rhs, instead of standard equality?:

```
instantiation
  bit0 and bit1 :: (finite) equal
begin

definition equal-bit0 :: 'a bit0 => 'a bit0 => bool
  where equal-bit0 x y = (x = y)

definition equal-bit1 :: 'a bit1 => 'a bit1 => bool
  where equal-bit1 x y = (x = y)

instance
  by (intro-classes, unfold equal-bit0-def equal-bit1-def)
    (metis bit0.Rep-inverse, metis bit1.Rep-inverse)

end
```

The previous definitions of *equal-class.equal ?x ?y* = (*?x* = *?y*) and *equal-class.equal ?x ?y* = (*?x* = *?y*) would make the code generator loop; instead, we introduce the following ones:

```
lemma equal-bit0-code [code]:
  equal-class.equal x y = (Rep-bit0 x = Rep-bit0 y)
  by (simp add: equal-eq Rep-bit0-inject)

lemma equal-bit1-code [code]:
  equal-class.equal x y = (Rep-bit1 x = Rep-bit1 y)
  by (simp add: equal-eq Rep-bit1-inject)

lemma card-1 [simp]: card {1::num1} = 1 by simp
```

The definitions of *equal-class.equal ?x ?y* = (*?x* = *?y*) and *equal-class.equal ?x ?y* = (*?x* = *?y*) preserve the equality of elements:

```
lemma equal-class.equal (2::4) (6::4)
  using equal-bit0-def [of 2::4 6] by auto

lemma equal-class.equal (2::5) (7::5)
  using equal-bit1-def [of 2::5 7] by auto
```

The type *0* is an instance of the classes *finite-UNIV* and *card-UNIV*, but with cardinal zero so it is not finite itself:

```
instantiation num0 :: card-UNIV begin

definition finite-UNIV = Phantom(num0) False

definition card-UNIV = Phantom(num0) 0
```

instance apply (*intro-classes*, *unfold finite-UNIV-num0-def card-UNIV-num0-def*)

apply (*metis* (*full-types*) *card-num0 finite-UNIV-card-ge-0 less-nat-zero-code*)
by (*metis* (*hide-lams*, *mono-tags*) *card-num0*)

end

The type *1* with a single element is an instance of the classes *finite-UNIV* and *card-UNIV*, with cardinal equal to one:

instantiation *num1* :: *card-UNIV* **begin**

definition *finite-UNIV* = *Phantom*(*num1*) *True*

definition *card-UNIV* = *Phantom*(*num1*) *1*

instance apply (*intro-classes*, *unfold finite-UNIV-num1-def card-UNIV-num1-def*)
by (*metis* (*full-types*) *finite-code*, *metis card-num1*)

end

The two following lemmas are required to prove that *'a bit0* and *'a bit1* are instances of *finite-UNIV*; under the assumption of the underlying type *'a* of being finite, they are easy to prove; otherwise, I do not know how to prove them; the premise about classes being instance of finite then follows along the development:

lemma *finite-bit0-UNIV-iff*: *finite* (*UNIV*::*'a*::*finite set*) ⟷ *finite* (*UNIV*::*'a bit0 set*)
by (*metis finite-class.finite-UNIV*)

lemma *finite-bit1-UNIV-iff*: *finite* (*UNIV*::*'a*::*finite set*) ⟷ *finite* (*UNIV*::*'a bit1 set*)
by (*metis finite-class.finite-UNIV*)

With the previous results, now we can prove that *'a bit0* is an instance of the type class *finite-UNIV* and that *'a bit0* is an instance of the type class *card-UNIV*:

instantiation *bit0* :: ({*finite*,*finite-UNIV*}) *finite-UNIV* **begin**

definition *finite-UNIV* = *Phantom*(*'a bit0*) (*of-phantom* (*finite-UNIV* :: *'a finite-UNIV*))

instance
by (*intro-classes*, *unfold finite-UNIV-bit0-def finite-UNIV-code* [*symmetric*], *rule cong* [*of Phantom*(*'a bit0*)], *rule refl*) (*rule finite-bit0-UNIV-iff*)

end

instantiation *bit0* :: ({*finite*,*card-UNIV*}) *card-UNIV* **begin**

definition *card-UNIV* = *Phantom*(*'a bit0*) (*2* ∗ *of-phantom* (*card-UNIV* :: *'a card-UNIV*))

instance
by (*intro-classes*, *unfold card-UNIV-bit0-def finite-UNIV-bit0-def card-UNIV-code* [*symmetric*])
(*rule cong* [*of Phantom* (*'a bit0*)], *rule refl*, *rule card-bit0* [*symmetric*])

end

With the previous results, now we can prove that *'a bit1* is an instance of the type class *finite-UNIV* and that *'a bit1* is an instance of the type class *card-UNIV*:

instantiation *bit1* :: ({*finite*,*finite-UNIV*}) *finite-UNIV* **begin**

definition *finite-UNIV* = *Phantom*(*'a bit1*) (*of-phantom* (*finite-UNIV* :: *'a finite-UNIV*))

instance by (*intro-classes*, *unfold finite-UNIV-bit1-def finite-UNIV-code* [*symmetric*],

rule cong [*of Phantom*(*'a bit1*)], *rule refl*) (*rule finite-bit1-UNIV-iff*)

end

instantiation *bit1* :: ({*finite*,*card-UNIV*}) *card-UNIV* **begin**

definition *card-UNIV* = *Phantom*(*'a bit1*) (*Suc* (*2* ∗ *of-phantom* (*card-UNIV* :: *'a card-UNIV*)))

instance
by (*intro-classes*, *unfold card-UNIV-bit1-def finite-UNIV-bit1-def card-UNIV-code* [*symmetric*],
rule cong [*of Phantom* (*'a bit1*)], *rule refl*, *rule card-bit1* [*symmetric*])

end

Some other interesting instantiations that are missed in the Library:

instantiation *bit0* **and** *bit1* :: (*finite*) *linorder* **begin**

definition *less-bit0* :: *'a bit0* => *'a bit0* => *bool*
where *less-bit0 a b* == (*Rep-bit0 a*) < (*Rep-bit0 b*)

definition *less-eq-bit0* :: *'a bit0* => *'a bit0* => *bool*
where *less-eq-bit0 a b* == (*Rep-bit0 a*) ≤ (*Rep-bit0 b*)

definition *less-bit1* :: *'a bit1* => *'a bit1* => *bool*
where *less-bit1 a b* == (*Rep-bit1 a*) < (*Rep-bit1 b*)

definition *less-eq-bit1* :: *'a bit1* => *'a bit1* => *bool*
where *less-eq-bit1 a b* == (*Rep-bit1 a*) ≤ (*Rep-bit1 b*)

instance
  by (*intro-classes*, *unfold less-eq-bit0-def less-bit0-def*,
    unfold less-eq-bit1-def less-bit1-def, *auto*)
   (*metis bit0.Rep-inverse*, *metis bit1.Rep-inverse*)

end

instantiation *num1* :: *enum* **begin**

definition (*enum-class.enum*::*num1 list*) = [*1*::*1*]

definition *enum-class.enum-all* (*P*::*num1* ⇒ *bool*) = *P 1*

definition *enum-class.enum-ex* (*P*::*num1* ⇒ *bool*) = *P 1*

instance proof
  show (*UNIV*::*num1 set*) = *set enum-class.enum*
    unfolding *enum-num1-def* **using** *num1-eq-iff* **by** *fastforce*
  show *distinct* (*enum-class.enum*::*num1 list*)
    unfolding *enum-num1-def* **using** *num1-eq-iff* **by** *fastforce*
  fix *P*::*1* ⇒ *bool* **show** *enum-class.enum-all P* = *Ball UNIV P*
    unfolding *enum-all-num1-def* **unfolding** *enum-num1-def Ball-def*
    using *num1-eq-iff* **by** (*auto*)
  show *enum-class.enum-ex P* = *Bex UNIV P*
    unfolding *enum-ex-num1-def enum-num1-def Ball-def*
    using *num1-eq-iff* **by** (*auto*)
qed

end

We are now to prove that *'a bit0* and *'a bit1* are instances of the type class enum; we must provide an enumeration of the universe of that types; we first provide an explicit expression of such lists, by means of the following mappings:

definition *create-list-bit0* :: *nat* => (*'b*::{*finite*} *bit0*) *list*
  where *create-list-bit0 n* = *map* ((*Abs-bit0'*::*int* ⇒ *'b*::*finite bit0*) ∘ *int*) (*upt 0 n*)

definition *create-list-bit1* :: *nat* => (*'b*::{*finite*} *bit1*) *list*
  where *create-list-bit1 n* = *map* ((*Abs-bit1'*::*int* ⇒ *'b*::*finite bit1*) ∘ *int*) (*upt 0 n*)

Some relevant properties of *create-list-bit0 ?n* = *map* (*Abs-bit0'* ∘ *int*) [*0*..<*?n*]:

lemma *create-list-bit0-0*: *create-list-bit0* (*0*::*nat*) = []
  unfolding *create-list-bit0-def* **by** *auto*

lemma *length-create-list-bit0*: *length* (*create-list-bit0* (*n*)) = *n*

unfolding *create-list-bit0-def* **unfolding** *length-map* **by** *auto*

We have to prove that the elements inside of the lit are different of each other, as long as the index n is smaller than the cardinal of the underlying type

lemma *distinct-create-list-bit0*:
assumes *n*: $n \leq 2{*}CARD('a::finite)$
shows *distinct* ((*create-list-bit0 n*)::'*a bit0 list*)
proof (*unfold create-list-bit0-def distinct-map inj-on-def*, *auto*)
fix *x y*::*nat* **assume** *x*: $x < n$ **and** *y*: $y < n$
and *Abs-eq*: (*Abs-bit0' (int x)*::'*a bit0*) = *Abs-bit0' (int y)*
have *Abs-bit0 (int y) = Abs-bit0 (int (y mod CARD ('a bit0)))* **using** *mod-less y n* **by** *auto*
also have ... = *Abs-bit0 (int y mod int CARD('a bit0))* **unfolding** *transfer-int-nat-functions(2)* ..
also have ... = (*Abs-bit0' (int y)*::'*a bit0*) **using** *Abs-bit0'-def*[*of int y*, **where** *?'a='a*] ..
also have ... = (*Abs-bit0' (int x)*::'*a bit0*) **using** *Abs-eq* ..
also have ... = *Abs-bit0 (int x mod int CARD('a bit0))*
using *Abs-bit0'-def*[*of int x*, **where** *?'a='a*] .
also have ... = *Abs-bit0 (int (x mod CARD ('a bit0)))* **unfolding** *transfer-int-nat-functions(2)* ..
also have ... = *Abs-bit0 (int (x))* **using** *mod-less x n* **by** *auto*
finally have *Abs-eq'*: (*Abs-bit0 (int (y::nat))*::'*a bit0*) = *Abs-bit0 (int (x::nat))* **by** *auto*
have ((*Abs-bit0 (int (y::nat))*::'*a bit0*) = *Abs-bit0 (int (x::nat))*) = (*int y = int x*)
proof (*rule Abs-bit0-inject*)
have $x \in \{0..<(2 * CARD('a))\}$ **using** *x n* **by** *auto*
thus $int\ x \in \{0::int..<(2::int) * int\ CARD('a)\}$ **by** *auto*
have $y \in \{0..<(2 * CARD('a))\}$ **using** *y n* **by** *auto*
thus $int\ y \in \{0::int..<(2::int) * int\ CARD('a)\}$ **by** *auto*
qed
thus *x=y* **using** *Abs-eq'* **by** *simp*
qed

corollary *distinct-create-list-bit0'*:
assumes *n*: $n \leq CARD('a::finite\ bit0)$
shows *distinct* ((*create-list-bit0 n*)::'*a bit0 list*)
by (*rule distinct-create-list-bit0*) (*unfold card-bit0* [*symmetric*], *rule n*)

Since all the elements in the list are different, the set to which the list gives place has equal cardinal:

lemma *card-create-list-bit0*:
assumes *n*:$n \leq 2{*}\ CARD\ ('a::finite)$
shows *card (set ((create-list-bit0 (n))::'a bit0 list)) = n*
using *distinct-card*[*OF distinct-create-list-bit0*[*OF n*]] **unfolding** *length-create-list-bit0*
.

```
corollary card-create-list-bit0':
  assumes n: n ≤ CARD ('a::finite bit0)
  shows card (set ((create-list-bit0 n)::'a bit0 list)) = n
  by (rule card-create-list-bit0) (unfold card-bit0 [symmetric], rule n)
```

Some relevant properties of *create-list-bit1 ?n = map (Abs-bit1' ∘ int) [0..<?n]*

```
lemma create-list-bit1-0: create-list-bit1 (0::nat) = []
  unfolding create-list-bit1-def by auto

lemma length-create-list-bit1: length (create-list-bit1 (n)) = n
  unfolding create-list-bit1-def unfolding length-map by auto

lemma distinct-create-list-bit1:
  assumes n: n ≤ 1 + 2 ∗ CARD('a::finite)
  shows distinct ((create-list-bit1 (n))::'a bit1 list)
proof (unfold create-list-bit1-def distinct-map inj-on-def, auto)
  fix x y::nat assume x: x < n and y: y < n
    and Abs-eq: (Abs-bit1' (int x)::'a bit1) = Abs-bit1' (int y)
  have (Abs-bit1 (int y)::'a bit1) = Abs-bit1 (int (y mod CARD ('a bit1)))
    using mod-less[of y CARD ('a)] using y n by auto
 also have ... = Abs-bit1 (int y mod int CARD('a bit1)) unfolding transfer-int-nat-functions(2)
..
  also have ... = (Abs-bit1' (int y)::'a bit1) using Abs-bit1'-def[of int y, where
?'a='a] ..
  also have ... = (Abs-bit1' (int x)::'a bit1) using Abs-eq ..
  also have ... = Abs-bit1 (int x mod int CARD('a bit1))
    using Abs-bit1'-def[of int x, where ?'a='a] .
 also have ... = Abs-bit1 (int (x mod CARD ('a bit1))) unfolding transfer-int-nat-functions(2)
..
  also have ... = Abs-bit1 (int (x)) using mod-less x n by auto
  finally have Abs-eq': (Abs-bit1 (int (y::nat))::'a bit1) = Abs-bit1 (int (x::nat))
by auto
  have ((Abs-bit1 (int (y::nat))::'a bit1) = Abs-bit1 (int (x::nat))) = (int y = int
x)
  proof (rule Abs-bit1-inject)
    have x ∈ {0..< 1 + (2 ∗ CARD('a))} using x n by auto
    thus int x ∈ {0::int..<1 + (2::int) ∗ int CARD('a)} by auto
    have y ∈ {0..< 1 +(2 ∗ CARD('a))} using y n by auto
    thus int y ∈ {0::int..< 1 + (2::int) ∗ int CARD('a)} by auto
  qed
  thus x=y using Abs-eq' by simp
qed

corollary distinct-create-list-bit1':
  assumes n: n ≤ CARD('a::finite bit1)
  shows distinct ((create-list-bit1 n)::'a bit1 list)
  by (rule distinct-create-list-bit1)
    (unfold Suc-eq-plus1-left [symmetric], unfold card-bit1 [symmetric], rule n)
```

lemma *card-create-list-bit1*:
  assumes *n*:$n \leq 1 + 2 * CARD\ ('a::finite)$
  shows *card* (*set* ((*create-list-bit1* (*n*))::$'a$ *bit1 list*)) = *n*
 using *distinct-card*[*OF distinct-create-list-bit1* [*OF n*]] **unfolding** *length-create-list-bit1*
.

corollary *card-create-list-bit1'*:
  assumes *n*: $n \leq CARD\ ('a::finite\ bit1)$
  shows *card* (*set* ((*create-list-bit1 n*)::$'a$ *bit1 list*)) = *n*
  by (*rule card-create-list-bit1*)
    (*unfold Suc-eq-plus1-left* [*symmetric*], *unfold card-bit1* [*symmetric*], *rule n*)

The type constructors bit0 and bit1, over finite types, are instances of type class enum; we make use of the previously defined functions *create-list-bit0* *?n* = *map* (*Abs-bit0'* ∘ *int*) [*0..<?n*] and *create-list-bit1 ?n* = *map* (*Abs-bit1'* ∘ *int*) [*0..<?n*]:

instantiation *bit0* :: (*finite*) *enum*
begin

definition (*enum-class.enum*::$'a$ *bit0 list*) = *create-list-bit0* (*CARD*($'a$ *bit0*))
definition *enum-class.enum-all* (*P*::$'a$ *bit0* ⇒ *bool*) = (*filter P enum-class.enum* = *enum-class.enum*)
definition *enum-class.enum-ex* (*P*::$'a$ *bit0* ⇒ *bool*) = (*filter P enum-class.enum* ≠ [])

instance proof (*default*, *unfold ball-UNIV bex-UNIV*)
  show *univ-eq*:(*UNIV*::$'a$ *bit0 set*) = *set enum-class.enum*
  proof (*rule card-eq-UNIV-imp-eq-UNIV*[*symmetric*])
    show *finite* (*UNIV*::$'a$ *bit0 set*) **by** (*metis finite-class.finite-UNIV*)
    show *card* (*set* (*enum-class.enum*::$'a$ *bit0 list*)) = *CARD*($'a$ *bit0*)
    unfolding *enum-bit0-def* **using** *card-create-list-bit0* **by** *auto*
   qed
  show *distinct* (*enum-class.enum*::$'a$ *bit0 list*)
    unfolding *enum-bit0-def* **by** (*metis distinct-create-list-bit0' order-refl*)
  fix *P* :: $'a$ *bit0* ⇒ *bool*
  show *enum-class.enum-all P* = *All P*
    unfolding *enum-all-bit0-def*
    unfolding *filter-id-conv* **unfolding** *univ-eq*[*symmetric*] **by** *simp*
  show *enum-class.enum-ex P* = *Ex P*
    unfolding *enum-ex-bit0-def*
    unfolding *filter-empty-conv* **unfolding** *univ-eq*[*symmetric*] **by** *fast*
qed

end

The following restriction in the type class of the type $'a$ is necessary to get code generation; not only the underlying type $'a$ has to be finite, but also of the type class *card-UNIV*, which is the only one which knows how to compute the cardinal of types; more explanations in the mail by Andreas Lochbihler:

lemmas *enum-bit0-code* [*code*] = *enum-bit0-def* [**where** *?'a='a* :: {*finite*, *card-UNIV*}]

instantiation *bit1* :: (*finite*) *enum* **begin**

definition (*enum-class.enum*::*'a bit1 list*) = *create-list-bit1* (*CARD*(*'a bit1*))
definition *enum-class.enum-all* (*P*::*'a bit1* ⇒ *bool*) = (*filter P enum-class.enum* = *enum-class.enum*)
definition *enum-class.enum-ex* (*P*::*'a bit1* ⇒ *bool*) = (*filter P enum-class.enum* ≠ [])

instance proof (*default*, *unfold ball-UNIV bex-UNIV*)
  show *univ-eq*: (*UNIV*::*'a bit1 set*) = *set enum-class.enum*
  proof (*rule card-eq-UNIV-imp-eq-UNIV*[*symmetric*])
    show *finite* (*UNIV*::*'a bit1 set*) **by** (*metis finite*)
    show *card* (*set* (*enum-class.enum*::*'a bit1 list*)) = *CARD*(*'a bit1*)
      unfolding *enum-bit1-def* **using** *card-create-list-bit1*[*of 2*∗*CARD*(*'a*)+*1*, **where** *?'a='a*] **by** *simp*
  qed
  show *distinct* (*enum-class.enum*::*'a bit1 list*)
    unfolding *enum-bit1-def* **by** (*metis distinct-create-list-bit1' le-refl*)
  fix *P* :: *'a bit1* ⇒ *bool*
  show *enum-class.enum-all P* = *All P*
    unfolding *enum-all-bit1-def univ-eq*[*symmetric*] *filter-id-conv* **by** *fast*
  show *enum-class.enum-ex P* = *Ex P*
    unfolding *enum-ex-bit1-def*
    unfolding *filter-empty-conv* **unfolding** *univ-eq*[*symmetric*] **by** *simp*
qed
end

instantiation *vec* :: (*type*, *finite*) *equal*
begin
definition *equal-vec* :: (*'a*, *'b*::*finite*) *vec* => (*'a*, *'b*::*finite*) *vec* => *bool*
  where *equal-vec x y* = (∀ *i*. *x*$*i* = *y*$*i*)
instance
proof (*intro-classes*)
  fix *x y*::(*'a*, *'b*::*finite*) *vec*
  show *equal-class.equal x y* = (*x* = *y*) **unfolding** *equal-vec-def* **using** *vec-eq-iff* **by** *auto*
qed
end

'a bit0 and *'a bit1* are instances of the type class wellorder:

lemma (**in** *preorder*) *tranclp-less*: *op* $<^{++}$ = *op* <
 by(*auto simp add*: *fun-eq-iff intro*: *less-trans elim*: *tranclp.induct*)

instance *bit0* **and** *bit1* :: (*finite*) *wellorder*
proof −
have *wf* $\{(x :: 'a\ bit0,\ y).\ x < y\}$
by(*auto simp add*: *trancl-def tranclp-less intro*!: *finite-acyclic-wf acyclicI*)
thus *OFCLASS*('*a bit0*, *wellorder-class*)
by(*rule wf-wellorderI*) *intro-classes*
next
have *wf* $\{(x :: 'a\ bit1,\ y).\ x < y\}$
by(*auto simp add*: *trancl-def tranclp-less intro*!: *finite-acyclic-wf acyclicI*)
thus *OFCLASS*('*a bit1*, *wellorder-class*)
by(*rule wf-wellorderI*) *intro-classes*
qed

The following restriction in the type class of the type $'a$ is necessary to get code generation; not only the underlying type $'a$ has to be finite, but also of the type class *card-UNIV*, which is the only one which knows how to compute the cardinal of types; more explanations in the mail by Andreas Lochbihler:

lemmas *enum-bit1-code* [*code*] = *enum-bit1-def*[**where** *?'a='a* :: {*finite*, *card-UNIV*}]

end

theory *Dual-Order*
imports *Main*
begin

Computable Greatest value operator for finite linorder classes. Based on *Least* $?P = (THE\ x.\ ?P\ x \wedge (\forall y.\ ?P\ y \longrightarrow x \leq y))$

interpretation *dual-order*: *order* $(op \geq)$::('*a*::{*order*}=>'*a*=>*bool*) $(op >)$
proof
fix $x\ y$::'*a*::{*order*} **show** $(y < x) = (y \leq x \wedge \neg\ x \leq y)$ **using** *less-le-not-le* **.**
show $x \leq x$ **using** *order-refl* **.**
fix z **show** $y \leq x \Longrightarrow z \leq y \Longrightarrow z \leq x$ **using** *order-trans* **.**
next
fix $x\ y$::'*a*::{*order*} **show** $y \leq x \Longrightarrow x \leq y \Longrightarrow x = y$ **by** (*metis eq-iff*)
qed

interpretation *dual-linorder*: *linorder* $(op \geq)$::('*a*::{*linorder*}=>'*a*=>*bool*) $(op >)$
proof
fix $x\ y$::'*a* **show** $y \leq x \vee x \leq y$ **using** *linear* **.**
qed

lemma *wf-wellorderI2*:
assumes *wf*: *wf* $\{(x::'a::ord,\ y).\ y < x\}$
assumes *lin*: *class.linorder* $(\lambda(x::'a)\ y::'a.\ y \leq x)$ $(\lambda(x::'a)\ y::'a.\ y < x)$

shows *class.wellorder* $(\lambda(x::'a)\ y::'a.\ y \le x)$ $(\lambda(x::'a)\ y::'a.\ y < x)$
using *lin* **unfolding** *class.wellorder-def* **apply** (*rule conjI*)
apply (*rule class.wellorder-axioms.intro*) **by** (*blast intro*: *wf-induct-rule* [*OF wf*])

lemma (**in** *preorder*) *tranclp-less'*: $op >^{++} = op >$
by(*auto simp add*: *fun-eq-iff intro*: *less-trans elim*: *tranclp.induct*)

interpretation *dual-wellorder*: *wellorder* $(op \ge)$::(*'a*::{*linorder*, *finite*}=>*'a*=>*bool*) $(op >)$
proof (*rule wf-wellorderI2*)
show *wf* $\{(x :: 'a,\ y).\ y < x\}$
by(*auto simp add*: *trancl-def tranclp-less' intro*!: *finite-acyclic-wf acyclicI*)
show *class.linorder* $(\lambda(x::'a)\ y::'a.\ y \le x)$ $(\lambda(x::'a)\ y::'a.\ y < x)$
unfolding *class.linorder-def* **unfolding** *class.linorder-axioms-def* **unfolding** *class.order-def*
unfolding *class.preorder-def* **unfolding** *class.order-axioms-def* **by** *auto*
qed

definition *Greatest'* :: (*'a*::*order* ⇒ *bool*) ⇒ *'a*::*order* (**binder** *GREATEST'* *10*)
where *Greatest'* *P* = *dual-order.Least P*

The following THE will be computable when the underlying type belongs to a suitable class (for example, Enum).

lemma [*code*]: *Greatest'* *P* = (*THE x*::*'a*::*order*. $P\ x \land (\forall\, y$::*'a*::*order*. $P\ y \longrightarrow y \le x))$
unfolding *Greatest'-def ord.Least-def* **by** *fastforce*

lemmas *Greatest'I2-order* = *dual-order.LeastI2-order*[*folded Greatest'-def*]
lemmas *Greatest'-equality* = *dual-order.Least-equality*[*folded Greatest'-def*]
lemmas *Greatest'I* = *dual-wellorder.LeastI*[*folded Greatest'-def*]
lemmas *Greatest'I2-ex* = *dual-wellorder.LeastI2-ex*[*folded Greatest'-def*]
lemmas *Greatest'I2-wellorder* = *dual-wellorder.LeastI2-wellorder*[*folded Greatest'-def*]
lemmas *Greatest'I-ex* = *dual-wellorder.LeastI-ex*[*folded Greatest'-def*]
lemmas *not-greater-Greatest'* = *dual-wellorder.not-less-Least*[*folded Greatest'-def*]
lemmas *Greatest'I2* = *dual-wellorder.LeastI2*[*folded Greatest'-def*]
lemmas *Greatest'-ge* = *dual-wellorder.Least-le*[*folded Greatest'-def*]

end

theory *Mod-Type*
imports
Numeral-Type-Addenda
Dual-Order
begin

Class for modular arithmetic. It is inspired by the locale mod_type.

class *mod-type* = *times* + *wellorder* + *neg-numeral* +

fixes *Rep* :: *'a => int*
and *Abs* :: *int => 'a*
assumes *type*: *type-definition Rep Abs* {*0..<int CARD* (*'a*)}
and *size1*: *1* < *int CARD* (*'a*)
and *zero-def*: *0* = *Abs 0*
and *one-def*: *1* = *Abs 1*
and *add-def*: *x* + *y* = *Abs* ((*Rep x* + *Rep y*) *mod* (*int CARD* (*'a*)))
and *mult-def*: *x* ∗ *y* = *Abs* ((*Rep x* ∗ *Rep y*) *mod* (*int CARD* (*'a*)))
and *diff-def*: *x* − *y* = *Abs* ((*Rep x* − *Rep y*) *mod* (*int CARD* (*'a*)))
and *minus-def*: − *x* = *Abs* ((− *Rep x*) *mod* (*int CARD* (*'a*)))
and *strict-mono-Rep*: *strict-mono Rep*
begin

lemma *size0*: *0* < *int CARD* (*'a*)
using *size1* **by** *simp*

lemmas *definitions* =
zero-def one-def add-def mult-def minus-def diff-def

lemma *Rep-less-n*: *Rep x* < *int CARD* (*'a*)
by (*rule type-definition.Rep* [*OF type*, *simplified*, *THEN conjunct2*])

lemma *Rep-le-n*: *Rep x* ≤ *int CARD* (*'a*)
by (*rule Rep-less-n* [*THEN order-less-imp-le*])

lemma *Rep-inject-sym*: *x* = *y* ⟷ *Rep x* = *Rep y*
by (*rule type-definition.Rep-inject* [*OF type*, *symmetric*])

lemma *Rep-inverse*: *Abs* (*Rep x*) = *x*
by (*rule type-definition.Rep-inverse* [*OF type*])

lemma *Abs-inverse*: *m* ∈ {*0..<int CARD* (*'a*)} ⟹ *Rep* (*Abs m*) = *m*
by (*rule type-definition.Abs-inverse* [*OF type*])

lemma *Rep-Abs-mod*: *Rep* (*Abs* (*m mod int CARD* (*'a*))) = *m mod int CARD* (*'a*)
by (*simp add*: *Abs-inverse pos-mod-conj* [*OF size0*])

lemma *Rep-Abs-0*: *Rep* (*Abs 0*) = *0*
apply (*rule Abs-inverse* [*of 0*])
using *size0* **by** *simp*

lemma *Rep-0*: *Rep 0* = *0*
by (*simp add*: *zero-def Rep-Abs-0*)

lemma *Rep-Abs-1*: *Rep* (*Abs 1*) = *1*
by (*simp add*: *Abs-inverse size1*)

lemma *Rep-1*: *Rep 1* = *1*

by (*simp add*: *one-def Rep-Abs-1*)

lemma *Rep-mod*: *Rep x mod int CARD* (*'a*) = *Rep x*
apply (*rule-tac x*=*x* **in** *type-definition.Abs-cases* [*OF type*])
apply (*simp add*: *type-definition.Abs-inverse* [*OF type*])
apply (*simp add*: *mod-pos-pos-trivial*)
done

lemmas *Rep-simps* =
Rep-inject-sym Rep-inverse Rep-Abs-mod Rep-mod Rep-Abs-0 Rep-Abs-1

Definitions to make transformations among elements of a modular class and naturals

definition *to-nat* :: *'a* => *nat*
where *to-nat* = *nat* ∘ *Rep*

definition *Abs'* :: *int* => *'a*
where *Abs' x* = *Abs*(*x mod int CARD* (*'a*))

definition *from-nat* :: *nat* ⇒ *'a*
where *from-nat* = (*Abs'* ∘ *int*)

lemma *bij-Rep*: *bij-betw* (*Rep*) (*UNIV*::*'a set*) {*0*..<*int CARD*(*'a*)}
proof (*unfold bij-betw-def*, *rule conjI*)
show *inj Rep* **by** (*metis strict-mono-imp-inj-on strict-mono-Rep*)
show *range Rep* = {*0*..<*int CARD*(*'a*)} **using** *Typedef.type-definition.Rep-range*[*OF type*] **.**
qed

lemma *mono-Rep*: *mono Rep* **by** (*metis strict-mono-Rep strict-mono-mono*)

lemma *Rep-ge-0*: *0* ≤ *Rep x* **using** *bij-Rep* **unfolding** *bij-betw-def* **by** *auto*

lemma *bij-Abs*: *bij-betw* (*Abs*) {*0*..<*int CARD*(*'a*)} (*UNIV*::*'a set*)
proof (*unfold bij-betw-def*, *rule conjI*)
show *inj-on Abs* {*0*..<*int CARD*(*'a*)} **by** (*metis inj-on-inverseI type type-definition.Abs-inverse*)
show *Abs* ‘ {*0*..<*int CARD*(*'a*)} = (*UNIV*::*'a set*) **by** (*metis type type-definition.univ*)
qed

corollary *bij-Abs'*: *bij-betw* (*Abs'*) {*0*..<*int CARD*(*'a*)} (*UNIV*::*'a set*)
proof (*unfold bij-betw-def*, *rule conjI*)
show *inj-on Abs'* {*0*..<*int CARD*(*'a*)} **unfolding** *inj-on-def Abs'-def* **by** (*auto*, *metis Rep-Abs-mod mod-pos-pos-trivial*)
show *Abs'* ‘ {*0*..<*int CARD*(*'a*)} = (*UNIV*::*'a set*) **unfolding** *image-def Abs'-def*
apply *auto*
proof −
fix *x* **show** ∃ *xa*∈{*0*..<*int CARD*(*'a*)}. *x* = *Abs* (*xa mod int CARD*(*'a*))
by (*rule bexI*[*of - Rep x*], *auto simp add*: *Rep-less-n*[*of x*] *Rep-ge-0*[*of x*], *metis Rep-inverse Rep-mod*)

  qed
qed

lemma *bij-from-nat*: *bij-betw* (*from-nat*) {*0*..<*CARD*('*a*)} (*UNIV*::'*a set*)
proof (*unfold bij-betw-def*, *rule conjI*)
  have *set-eq*: {*0*::*int*..<*int CARD*('*a*)} = *int*‘ {*0*..<*CARD*('*a*)} **apply** (*auto*)
  proof −
    fix *x*::*int* **assume** *x1*: (*0*::*int*) ≤ *x* **and** *x2*: *x* < *int CARD*('*a*) **show** *x* ∈ *int*
‘ {*0*::*nat*..<*CARD*('*a*)}
    proof (*unfold image-def*, *auto*, *rule bexI*[*of* - *nat x*])
      show *x* = *int* (*nat x*) **using** *x1* **by** *auto*
      show *nat x* ∈ {*0*::*nat*..<*CARD*('*a*)} **using** *x1 x2* **by** *auto*
    qed
  qed
  show *inj-on* (*from-nat*::*nat*⇒'*a*) {*0*::*nat*..<*CARD*('*a*)}
  proof (*unfold from-nat-def* , *rule comp-inj-on*)
   show *inj-on int* {*0*::*nat*..<*CARD*('*a*)} **by** (*metis inj-of-nat subset-inj-on top-greatest*)
    show *inj-on* (*Abs*'::*int*=>'*a*) (*int* ‘ {*0*::*nat*..<*CARD*('*a*)}) **using** *bij-Abs* **un-
folding** *bij-betw-def set-eq*
       by (*metis* (*hide-lams*, *no-types*) *Abs*'-*def Abs-inverse Rep-inverse Rep-mod
inj-on-def set-eq*)
  qed
 show (*from-nat*::*nat*=>'*a*)‘ {*0*::*nat*..<*CARD*('*a*)} = *UNIV* **unfolding** *from-nat-def*
using *bij-Abs*' **unfolding** *bij-betw-def set-eq o-def* **by** *blast*
qed

lemma *to-nat-is-inv*: *the-inv-into* {*0*..<*CARD*('*a*)} (*from-nat*::*nat*=>'*a*) = (*to-nat*::'*a*=>*nat*)
proof (*unfold the-inv-into-def fun-eq-iff from-nat-def to-nat-def o-def*, *clarify*)
  fix *x*::'*a* **show** (*THE y*::*nat*. *y* ∈ {*0*::*nat*..<*CARD*('*a*)} ∧ *Abs*' (*int y*) = *x*) =
nat (*Rep x*)
  proof (*rule the-equality*, *auto*)
    show *Abs*' (*Rep x*) = *x* **by** (*metis Abs*'-*def Rep-inverse Rep-mod*)
    show *nat* (*Rep x*) < *CARD*('*a*) **by** (*metis* (*full-types*) *Rep-less-n nat-int size0
zless-nat-conj*)
    assume *x*: ¬ (*0*::*int*) ≤ *Rep x* **show** (*0*::*nat*) < *CARD*('*a*) **and** *Abs*' (*0*::*int*)
= *x* **using** *Rep-ge-0 x* **by** *auto*
  next
    fix *y*::*nat* **assume** *y*: *y* < *CARD*('*a*)
    have (*Rep*(*Abs*'(*int y*)::'*a*)) = (*Rep*((*Abs*(*int y mod int CARD*('*a*)))::'*a*)) **un-
folding** *Abs*'-*def* **..**
    also have ... = (*Rep* (*Abs* (*int y*)::'*a*)) **using** *zmod-int*[*of y CARD*('*a*)] **using**
y mod-less **by** *auto*
     also have ... = (*int y*) **proof** (*rule Abs-inverse*) **show** *int y* ∈ {*0*::*int*..<*int
CARD*('*a*)} **using** *y* **by** *auto* **qed**
    finally show *y* = *nat* (*Rep* (*Abs*' (*int y*)::'*a*)) **by** (*metis nat-int*)
  qed
qed

lemma *bij-to-nat*: *bij-betw* (*to-nat*) (*UNIV*::'*a set*) {*0*..<*CARD*('*a*)}

using *bij-betw-the-inv-into*[*OF bij-from-nat*] **unfolding** *to-nat-is-inv* **.**

lemma *finite-mod-type*: *finite* (*UNIV*::*'a set*)
using *finite-imageD*[*of to-nat UNIV*::*'a set*] **using** *bij-to-nat* **unfolding** *bij-betw-def*
by *auto*

subclass (**in** *mod-type*) *finite* **by** (*intro-classes*, *rule finite-mod-type*)

lemma *least-0*: (*LEAST n. n* ∈ (*UNIV*::*'a set*)) = *0*
proof (*rule Least-equality*, *auto*)
fix *y*::*'a*
have (*0*::*'a*) ≤ *Abs* (*Rep y mod int CARD*(*'a*)) **using** *strict-mono-Rep* **unfolding** *strict-mono-def*
by (*metis* (*hide-lams*, *mono-tags*) *Rep-0 Rep-ge-0 strict-mono-Rep strict-mono-less-eq*)
also have ... = *y* **by** (*metis Rep-inverse Rep-mod*)
finally show (*0*::*'a*) ≤ *y* **.**
qed

lemma *add-to-nat-def*: *x* + *y* = *from-nat* (*to-nat x* + *to-nat y*)
unfolding *from-nat-def to-nat-def o-def* **using** *Rep-ge-0*[*of x*] **using** *Rep-ge-0*[*of y*] **using** *Rep-less-n*[*of x*] *Rep-less-n*[*of y*]
unfolding *Abs'-def* **unfolding** *add-def*[*of x y*] **by** *auto*

lemma *to-nat-1*: *to-nat 1* = *1*
by (*metis* (*hide-lams*, *mono-tags*) *Rep-1 comp-apply to-nat-def transfer-nat-int-numerals*(*2*))

lemma *add-def'*:
shows *x* + *y* = *Abs'* (*Rep x* + *Rep y*) **unfolding** *Abs'-def* **using** *add-def* **by** *simp*

lemma *Abs'-0*:
shows *Abs'* (*CARD*(*'a*))=(*0*::*'a*) **by** (*metis* (*hide-lams*, *mono-tags*) *Abs'-def mod-self zero-def*)

lemma *Rep-plus-one-le-card*:
assumes *a*: *a* + *1* ≠ *0*
shows (*Rep a*) + *1* < *CARD* (*'a*)
proof (*rule ccontr*)
assume ¬ *Rep a* + *1* < *CARD*(*'a*) **hence** *to-nat-eq-card*: *Rep a* + *1* = *CARD*(*'a*)

by (*metis* (*hide-lams*, *mono-tags*) *Rep-less-n add1-zle-eq dual-order.le-less*)
have *a*+*1* = *Abs'* (*Rep a* + *Rep* (*1*::*'a*)) **using** *add-def'* **by** *auto*
also have ... = *Abs'* ((*Rep a*) + *1*) **using** *Rep-1* **by** *simp*
also have ... = *Abs'* (*CARD*(*'a*)) **unfolding** *to-nat-eq-card* **..**
also have ... = *0* **using** *Abs'-0* **by** *auto*
finally show *False* **using** *a* **by** *contradiction*
qed

lemma *to-nat-plus-one-less-card*: ∀ *a. a*+*1* ≠ *0* −−> *to-nat a* + *1* < *CARD*(*'a*)

proof (*clarify*)
fix *a*
assume *a*: *a* + *1* $\neq$ *0*
have *Rep a* + *1* < *int CARD*(*'a*) **using** *Rep-plus-one-le-card*[*OF a*] **by** *auto*
hence *nat* (*Rep a* + *1*) < *nat* (*int CARD*(*'a*)) **unfolding** *zless-nat-conj* **using** *size0* **by** *fast*
thus *to-nat a* + *1* < *CARD*(*'a*) **unfolding** *to-nat-def o-def* **using** *nat-add-distrib*[*OF Rep-ge-0*] **by** *simp*
qed

corollary *to-nat-plus-one-less-card'*:
assumes *a*+*1* $\neq$ *0*
shows *to-nat a* + *1* < *CARD*(*'a*) **using** *to-nat-plus-one-less-card assms* **by** *simp*

lemma *strict-mono-to-nat*: *strict-mono to-nat*
  using *strict-mono-Rep*
   unfolding *strict-mono-def to-nat-def* **using** *Rep-ge-0* **by** (*metis comp-apply nat-less-eq-zless*)

lemma *to-nat-eq* [*simp*]: *to-nat x* = *to-nat y* <−> *x* = *y* **using** *injD* [*OF bij-betw-imp-inj-on*[*OF bij-to-nat*]] **by** *blast*

lemma *mod-type-forall-eq* [*simp*]: ($\forall$ *j*::*'a*. (*to-nat j*)<*CARD*(*'a*) $\longrightarrow$ *P j*) = ($\forall$ *a*. *P a*)
proof (*auto*)
  fix *a* **assume** *a*: $\forall$ *j*. (*to-nat*::*'a*=>*nat*) *j* < *CARD*(*'a*) $\longrightarrow$ *P j*
  have (*to-nat*::*'a*=>*nat*) *a* < *CARD*(*'a*) **using** *bij-to-nat* **unfolding** *bij-betw-def* **by** *auto*
  thus *P a* **using** *a* **by** *auto*
qed

lemma *to-nat-from-nat*:
  assumes *t*:*to-nat j* = *k*
  shows *from-nat k* = *j*
proof −
  have *from-nat k* = *from-nat* (*to-nat j*) **unfolding** *t* **..**
  also have ... = *from-nat* (*the-inv-into* {*0*..<*CARD*(*'a*)} (*from-nat*) *j*) **unfolding** *to-nat-is-inv* **..**
  also have ... = *j*
  proof (*rule f-the-inv-into-f*)
   show *inj-on from-nat* {*0*..<*CARD*(*'a*)} **by** (*metis bij-betw-imp-inj-on bij-from-nat*)
   show *j* $\in$ *from-nat* ‘ {*0*..<*CARD*(*'a*)} **by** (*metis UNIV-I bij-betw-def bij-from-nat*)
  qed
  finally show *from-nat k* = *j* **.**
qed

lemma *to-nat-mono*:
  assumes *ab*: *a* < *b*
  shows *to-nat a* < *to-nat b*

```
  using strict-mono-to-nat unfolding strict-mono-def using assms by fast

lemma to-nat-mono':
  assumes ab: a ≤ b
  shows to-nat a ≤ to-nat b
proof (cases a=b)
  case True thus ?thesis by auto
next
  case False
  hence a<b using ab by simp
  thus ?thesis using to-nat-mono by fastforce
qed

lemma least-mod-type:
  shows 0 ≤ (n::'a)
  using least-0 by (metis (full-types) Least-le UNIV-I)

lemma to-nat-from-nat-id:
  assumes x: x<CARD('a)
  shows to-nat ((from-nat x)::'a) = x
  unfolding to-nat-is-inv[symmetric] proof (rule the-inv-into-f-f)
  show inj-on (from-nat::nat=>'a) {0..<CARD('a)} using bij-from-nat unfold-
ing bij-betw-def by auto
  show x ∈ {0..<CARD('a)} using x by simp
qed

lemma from-nat-to-nat-id[simp]:
  shows from-nat (to-nat x) = x by (metis to-nat-from-nat)

lemma from-nat-to-nat:
  assumes t:from-nat j = k and j: j<CARD('a)
  shows to-nat k = j by (metis j t to-nat-from-nat-id)

lemma from-nat-mono:
  assumes i-le-j: i<j and j: j<CARD('a)
  shows (from-nat i::'a) < from-nat j
proof -
have i: i<CARD('a) using i-le-j j by simp
obtain a where a: i=to-nat a using bij-to-nat unfolding bij-betw-def using i
to-nat-from-nat-id by metis
obtain b where b: j=to-nat b using bij-to-nat unfolding bij-betw-def using j
to-nat-from-nat-id by metis
show ?thesis by (metis a b from-nat-to-nat-id i-le-j strict-mono-less strict-mono-to-nat)
qed

lemma from-nat-mono':
  assumes i-le-j: i ≤ j and j<CARD ('a)
  shows (from-nat i::'a) ≤ from-nat j
proof (cases i=j)
```

case *True*
have (*from-nat i*::′*a*) = *from-nat j* **using** *True* **by** *simp*
thus *?thesis* **by** *simp*
next
case *False*
hence *i*<*j* **using** *i-le-j* **by** *simp*
thus *?thesis* **by** (*metis assms*(*2*) *from-nat-mono less-imp-le*)
qed

lemma *to-nat-suc*:
assumes *to-nat* (*x*)+*1* < *CARD* (′*a*)
shows *to-nat* (*x* + *1*::′*a*) = (*to-nat x*) + *1*
proof −
have (*x*::′*a*) + *1* = *from-nat* (*to-nat x* + *to-nat* (*1*::′*a*)) **unfolding** *add-to-nat-def* **..**
hence *to-nat* ((*x*::′*a*) + *1*) = *to-nat* (*from-nat* (*to-nat x* + *to-nat* (*1*::′*a*))::′*a*) **by** *presburger*
also have ... = *to-nat* (*from-nat* (*to-nat x* + *1*)::′*a*) **unfolding** *to-nat-1* **..**
also have ... = (*to-nat x* + *1*) **by** (*metis assms to-nat-from-nat-id*)
finally show *?thesis* **.**
qed

lemma *to-nat-le*:
assumes *y* < *from-nat k*
shows *to-nat y* < *k*
proof (*cases k*<*CARD*(′*a*))
case *True* **show** *?thesis* **by** (*metis* (*full-types*) *True* ‹*y* < *from-nat k*› *to-nat-from-nat-id to-nat-mono*)
next
case *False* **have** *to-nat y* < *CARD* (′*a*) **using** *bij-to-nat* **unfolding** *bij-betw-def* **by** *auto*
thus *?thesis* **using** *False* **by** *auto*
qed

lemma *le-Suc*:
assumes *ab*: *a* < (*b*::′*a*)
shows *a* + *1* ≤ *b*
proof −
have *a* + *1* = (*from-nat* (*to-nat* (*a* + *1*))::′*a*) **using** *from-nat-to-nat-id* [*of a*+*1*,*symmetric*] **.**
also have ... ≤ (*from-nat* (*to-nat* (*b*::′*a*))::′*a*)
proof (*rule from-nat-mono*′)
have *to-nat a* < *to-nat b* **using** *ab* **by** (*metis to-nat-mono*)
hence *to-nat a* + *1* ≤ *to-nat b* **by** *simp*
thus *to-nat b* < *CARD* (′*a*) **using** *bij-to-nat* **unfolding** *bij-betw-def* **by** *auto*
hence *to-nat a* + *1* < *CARD* (′*a*) **by** (*metis* ‹*to-nat a* + *1* ≤ *to-nat b*› *preorder-class.le-less-trans*)
thus *to-nat* (*a* + *1*) ≤ *to-nat b* **by** (*metis* ‹*to-nat a* + *1* ≤ *to-nat b*› *to-nat-suc*)
qed

```
  also have ... = b by (metis from-nat-to-nat-id)
  finally show a + (1::'a) ≤ b .
qed

lemma le-Suc':
assumes ab: a + 1 ≤ b
  and less-card: (to-nat a) + 1 < CARD ('a)
  shows a < b
proof -
  have a = (from-nat (to-nat a)::'a) using from-nat-to-nat-id [of a,symmetric] .
  also have ... < (from-nat (to-nat b)::'a)
  proof (rule from-nat-mono)
    show to-nat b < CARD('a) using bij-to-nat unfolding  bij-betw-def by auto
    have to-nat (a + 1) ≤  to-nat b using ab by (metis to-nat-mono')
    hence to-nat (a) + 1 ≤  to-nat b  using to-nat-suc[OF less-card] by auto
    thus to-nat a < to-nat b by simp
  qed
  finally show a < b by (metis to-nat-from-nat)
qed

lemma Suc-le:
  assumes less-card: (to-nat a) + 1 < CARD ('a)
  shows a < a + 1
proof -
  have (to-nat a) < (to-nat a) + 1 by simp
  hence (to-nat a) < to-nat (a + 1) by (metis less-card to-nat-suc)
  hence (from-nat (to-nat a)::'a) < from-nat (to-nat (a + 1))
    by (rule from-nat-mono, metis less-card to-nat-suc)
  thus a < a + 1 by (metis to-nat-from-nat)
qed

lemma Suc-le':
  fixes a::'a
  assumes a + 1 ≠ 0
  shows a < a + 1 using Suc-le to-nat-plus-one-less-card assms by blast

lemma from-nat-not-eq:
  assumes a-eq-to-nat: a ≠ to-nat b
  and a-less-card: a<CARD('a)
  shows from-nat a ≠ b
proof (rule ccontr)
  assume ¬ from-nat a ≠ b hence from-nat a = b by simp
  hence to-nat ((from-nat a)::'a) = to-nat b by auto
  thus False by (metis a-eq-to-nat a-less-card to-nat-from-nat-id)
qed

lemma Suc-less:
  fixes i::'a
  assumes i<j
```

and $i+1 \neq j$
shows $i+1<j$ **by** (*metis assms le-Suc le-neq-trans*)

lemma *Greatest-is-minus-1*: $\forall\, a::'a.\ a \leq -1$
proof (*clarify*)
fix $a::'a$
have *zero-ge-card-1*: $0 \leq int\ CARD('a) - 1$ **using** *size1* **by** *auto*
have *card-less*: $int\ CARD('a) - 1 < int\ CARD('a)$ **by** *auto*
have *not-zero*: $1\ mod\ int\ CARD('a) \neq 0$ **by** (*metis* (*hide-lams*, *mono-tags*) *Rep-Abs-1 Rep-mod zero-neq-one*)
have *int-card*: $int\ (CARD('a) - 1) = int\ CARD('a) - 1$ **using** *zdiff-int*[*of 1 CARD ('a)*] **using** *size1* **by** *simp*
have $a = Abs'\ (Rep\ a)$ **by** (*metis* (*hide-lams*, *mono-tags*) *Rep-0 add-0-right add-def' comm-monoid-add-class.add.right-neutral*)
also have ... $= Abs'\ (int\ (nat\ (Rep\ a)))$ **by** (*metis Rep-ge-0 int-nat-eq*)
also have ... $\leq Abs'\ (int\ (CARD('a) - 1))$
proof (*rule from-nat-mono'*[*unfolded from-nat-def o-def*, *of nat* (*Rep a*) *CARD*(*'a*) *− 1*])
show $nat\ (Rep\ a) \leq CARD('a) - 1$ **using** *Rep-less-n*
by (*metis* (*hide-lams*, *mono-tags*) *Rep-1 Rep-le-n dual-linorder.leD dual-linorder.le-less-linear of-nat-1 of-nat-diff zle-diff1-eq zle-int zless-nat-eq-int-zless*)
show $CARD('a) - 1 < CARD('a)$ **using** *finite-UNIV-card-ge-0 finite-mod-type* **by** *fastforce*
qed
also have ... $= -1$
unfolding *Abs'-def* **unfolding** *minus-def zmod-zminus1-eq-if* **unfolding** *Rep-1*

apply (*rule cong* [*of Abs*], *rule refl*)
unfolding *if-not-P* [*OF not-zero*]
unfolding *int-card*
unfolding *mod-pos-pos-trivial*[*OF zero-ge-card-1 card-less*]
using *mod-pos-pos-trivial*[*OF - size1*] **by** *presburger*
finally show $a \leq -1$ **by** *fastforce*
qed

lemma *a-eq-minus-1*: $\forall\, a::'a.\ a+1 = 0 \longrightarrow a = -1$
by (*metis add-neg-numeral-special*(*2*) *add-right-cancel sub-num-simps*(*1*))

lemma *forall-from-nat-rw*:
shows $(\forall\, x\in\{0..<CARD('a)\}.\ P\ (from\text{-}nat\ x::'a)) = (\forall\, x.\ P\ (from\text{-}nat\ x))$
proof (*auto*)
fix y **assume** $*$: $\forall\, x\in\{0..<CARD('a)\}.\ P\ (from\text{-}nat\ x)$
have $from\text{-}nat\ y \in (UNIV::'a\ set)$ **by** *auto*
from *this* **obtain** x **where** *x1*: $from\text{-}nat\ y = (from\text{-}nat\ x::'a)$ **and** *x2*: $x\in\{0..<CARD('a)\}$
using *bij-from-nat* **unfolding** *bij-betw-def*
by (*metis from-nat-to-nat-id rangeI the-inv-into-onto to-nat-is-inv*)
show $P\ (from\text{-}nat\ y::'a)$ **unfolding** *x1* **using** $*$ *x2* **by** *simp*

```
qed

lemma from-nat-eq-imp-eq:
  assumes f-eq: from-nat x = (from-nat xa::'a)
and x: x<CARD('a) and xa: xa<CARD('a)
  shows x=xa using assms from-nat-not-eq by metis

lemma to-nat-less-card:
  fixes j::'a
  shows to-nat j < CARD ('a)
  using bij-to-nat unfolding bij-betw-def by auto

lemma from-nat-0: from-nat 0 = 0
  unfolding from-nat-def o-def of-nat-0 Abs'-def mod-0 zero-def ..
lemma to-nat-0: to-nat 0 = 0 unfolding to-nat-def o-def Rep-0 nat-0 ..
lemma to-nat-eq-0: (to-nat x = 0) = (x = 0) using to-nat-0 to-nat-from-nat by
auto


lemma suc-not-zero:
  assumes to-nat a + 1 ≠ CARD('a)
  shows a+1 ≠ 0
proof (rule ccontr, simp)
  assume a-plus-one-zero: a + 1 = 0
  hence rep-eq-card: Rep a + 1 = CARD('a) using assms to-nat-0 Suc-eq-plus1
Suc-lessI Zero-not-Suc to-nat-less-card to-nat-suc by (metis (hide-lams, mono-tags))
  moreover have Rep a + 1 < CARD('a)
   using Abs'-0 Rep-1 Suc-eq-plus1 Suc-lessI Suc-neq-Zero add-def' assms rep-eq-card
to-nat-0 to-nat-less-card to-nat-suc by (metis (hide-lams, mono-tags))
  ultimately show False by fastforce
qed
end

instantiation bit0 and bit1:: (finite) mod-type
begin

definition (Rep::'a bit0 => int)  x = Rep-bit0 x
definition (Abs::int => 'a bit0) x = Abs-bit0' x

definition (Rep::'a bit1 => int)  x = Rep-bit1 x
definition (Abs::int => 'a bit1) x = Abs-bit1' x

instance
proof
 show (0::'a bit0) = Abs (0::int) unfolding Abs-bit0-def Abs-bit0'-def zero-bit0-def
by auto
  show (1::int) < int CARD('a bit0) by (metis bit0.size1)
  show type-definition (Rep::'a bit0 => int) (Abs:: int => 'a bit0) {0::int..<int
CARD('a bit0)}
```

proof (*unfold type-definition-def Rep-bit0-def* [*abs-def*] *Abs-bit0-def* [*abs-def*] *Abs-bit0′-def*, *intro conjI*)
show $\forall x::'a\ bit0.\ Rep\text{-}bit0\ x \in \{0::int..<int\ CARD('a\ bit0)\}$
unfolding *card-bit0* **unfolding** *int-mult*
using *Rep-bit0* [**where** *?′a = ′a*] **by** *simp*
show $\forall x::'a\ bit0.\ Abs\text{-}bit0\ (Rep\text{-}bit0\ x\ mod\ int\ CARD('a\ bit0)) = x$
by (*metis Rep-bit0-inverse bit0.Rep-mod*)
show $\forall y::int.\ y \in \{0::int..<int\ CARD('a\ bit0)\} \longrightarrow Rep\text{-}bit0\ ((Abs\text{-}bit0::int => 'a\ bit0)\ (y\ mod\ int\ CARD('a\ bit0))) = y$
by (*metis bit0.Abs-inverse bit0.Rep-mod*)
qed
show $(1::'a\ bit0) = Abs\ (1::int)$ **unfolding** *Abs-bit0-def Abs-bit0′-def one-bit0-def*

by (*metis bit0.of-nat-eq of-nat-1 one-bit0-def*)
fix $x\ y :: 'a\ bit0$
show $x + y = Abs\ ((Rep\ x + Rep\ y)\ mod\ int\ CARD('a\ bit0))$
unfolding *Abs-bit0-def Rep-bit0-def plus-bit0-def Abs-bit0′-def* **by** *fastforce*
show $x * y = Abs\ (Rep\ x * Rep\ y\ mod\ int\ CARD('a\ bit0))$
unfolding *Abs-bit0-def Rep-bit0-def times-bit0-def Abs-bit0′-def* **by** *fastforce*
show $x - y = Abs\ ((Rep\ x - Rep\ y)\ mod\ int\ CARD('a\ bit0))$
unfolding *Abs-bit0-def Rep-bit0-def minus-bit0-def Abs-bit0′-def* **by** *fastforce*
show $- x = Abs\ (- Rep\ x\ mod\ int\ CARD('a\ bit0))$
unfolding *Abs-bit0-def Rep-bit0-def uminus-bit0-def Abs-bit0′-def* **by** *fastforce*
show $(0::'a\ bit1) = Abs\ (0::int)$ **unfolding** *Abs-bit1-def Abs-bit1′-def zero-bit1-def* **by** *auto*
show $(1::int) < int\ CARD('a\ bit1)$ **by** (*metis bit1.size1*)
show $(1::'a\ bit1) = Abs\ (1::int)$ **unfolding** *Abs-bit1-def Abs-bit1′-def one-bit1-def*
by (*metis bit1.of-nat-eq of-nat-1 one-bit1-def*)
fix $x\ y :: 'a\ bit1$
show $x + y = Abs\ ((Rep\ x + Rep\ y)\ mod\ int\ CARD('a\ bit1))$
unfolding *Abs-bit1-def Abs-bit1′-def Rep-bit1-def plus-bit1-def* **by** *fastforce*
show $x * y = Abs\ (Rep\ x * Rep\ y\ mod\ int\ CARD('a\ bit1))$
unfolding *Abs-bit1-def Rep-bit1-def times-bit1-def Abs-bit1′-def* **by** *fastforce*
show $x - y = Abs\ ((Rep\ x - Rep\ y)\ mod\ int\ CARD('a\ bit1))$
unfolding *Abs-bit1-def Rep-bit1-def minus-bit1-def Abs-bit1′-def* **by** *fastforce*
show $- x = Abs\ (- Rep\ x\ mod\ int\ CARD('a\ bit1))$
unfolding *Abs-bit1-def Rep-bit1-def uminus-bit1-def Abs-bit1′-def* **by** *fastforce*
show $type\text{-}definition\ (Rep::'a\ bit1 => int)\ (Abs::\ int => 'a\ bit1)\ \{0::int..<int\ CARD('a\ bit1)\}$
proof (*unfold type-definition-def Rep-bit1-def* [*abs-def*] *Abs-bit1-def* [*abs-def*] *Abs-bit1′-def*, *intro conjI*)
show $\forall x::'a\ bit1.\ Rep\text{-}bit1\ x \in \{0::int..<int\ CARD('a\ bit1)\}$
unfolding *card-bit1*
unfolding *int-Suc int-mult*
using *Rep-bit1* [**where** *?′a = ′a*] **by** *simp*
show $\forall x::'a\ bit1.\ Abs\text{-}bit1\ (Rep\text{-}bit1\ x\ mod\ int\ CARD('a\ bit1)) = x$
by (*metis Rep-bit1-inverse bit1.Rep-mod*)
show $\forall y::int.\ y \in \{0::int..<int\ CARD('a\ bit1)\} \longrightarrow Rep\text{-}bit1\ ((Abs\text{-}bit1::int => 'a\ bit1)\ (y\ mod\ int\ CARD('a\ bit1))) = y$

by (*metis bit1.Abs-inverse bit1.Rep-mod*)
qed
show *strict-mono* (*Rep*::'*a bit0* => *int*) **unfolding** *strict-mono-def* **by** (*metis Rep-bit0-def less-bit0-def*)
show *strict-mono* (*Rep*::'*a bit1* => *int*) **unfolding** *strict-mono-def* **by** (*metis Rep-bit1-def less-bit1-def*)
qed
end

end

1 Rank Nullity Theorem of Linear Algebra

theory *Dim-Formula*
imports ~~/*src*/*HOL*/*Multivariate-Analysis*/*Multivariate-Analysis*
begin

1.1 Previous results

Linear dependency is a monotone property, based on the monotonocity of linear independence:

lemma *dependent-mono*:
assumes *d*:*dependent A*
and *A-in-B*: $A \subseteq B$
shows *dependent B*
using *independent-mono* [*OF* - *A-in-B*] *d* **by** *auto*

The negation of

$$dependent\ P = (\exists S\ u.\ finite\ S \wedge S \subseteq P \wedge (\exists v{\in}S.\ u\ v \neq 0 \wedge (\sum v{\in}S.\ u\ v *_R v) = (0::'a)))$$

produces the following result:

lemma *independent-explicit*:
$independent\ A = (\forall S \subseteq A.\ finite\ S \longrightarrow (\forall u.\ (\sum v{\in}S.\ u\ v *_R v) = 0 \longrightarrow (\forall v{\in}S.\ u\ v = 0)))$
unfolding *dependent-explicit* [*of A*] **by** (*simp add*: *disj-not2*)

A finite set A for which every of its linear combinations equal to zero requires every coefficient being zero, is independent:

lemma *independent-if-scalars-zero*:
assumes *fin-A*: *finite A*
and *sum*: $\forall f.\ (\sum x{\in}A.\ f\ x *_R x) = 0 \longrightarrow (\forall x \in A.\ f\ x = 0)$
shows *independent A*

proof (*unfold independent-explicit*, *clarify*)
  fix S v **and** u :: $'a \Rightarrow real$
  assume S: $S \subseteq A$ **and** v: $v \in S$
  let $?g = \lambda x.$ *if* $x \in S$ *then* $u\ x$ *else* 0
  have $(\sum v{\in}A.\ ?g\ v *_R v) = (\sum v{\in}S.\ u\ v *_R\ v)$
    using S *fin-A* **by** (*auto intro*!: *setsum-mono-zero-cong-right*)
  also assume $(\sum v{\in}S.\ u\ v *_R v) = 0$
  finally have $?g\ v = 0$ **using** v S *sum* **by** *force*
  thus $u\ v = 0$ **unfolding** *if-P*[*OF* v] **.**
qed

Given a finite independent set, a linear combination of its elements equal to zero is possible only if every coefficient is zero:

lemma *scalars-zero-if-independent*:
  assumes *fin-A*: *finite* A
  and *ind*: *independent* A
  and *sum*: $(\sum x{\in}A.\ f\ x *_R x) = 0$
  shows $\forall\, x \in A.\ f\ x = 0$
  using *assms* **unfolding** *independent-explicit* **by** *auto*

In an euclidean space, every set is finite, and thus

$[\![$*finite* A; *independent* A; $(\sum x{\in}A.\ f\ x *_R x) = (0{::}'a)]\!] \Longrightarrow \forall\, x{\in}A.\ f\ x = 0$

holds:

corollary *scalars-zero-if-independent-euclidean*:
  fixes $A{::}'a{::}$*euclidean-space set*
  assumes *ind*: *independent* A
  and *sum*: $(\sum x{\in}A.\ f\ x *_R x) = 0$
  shows $\forall\, x \in A.\ f\ x = 0$
  by (*rule scalars-zero-if-independent*,
      rule conjunct1 [*OF independent-bound* [*OF ind*]])
    (*rule ind*, *rule sum*)

The following lemma states that every linear form is injective over the elements which define the basis of the range of the linear form. This property is applied later over the elements of an arbitrary basis which are not in the basis of the nullifier or kernel set (*i.e.*, the candidates to be the basis of the range space of the linear form).

Thanks to this result, it can be concluded that the cardinal of the elements of a basis which do not belong to the kernel of a linear form f is equal to the cardinal of the set obtained when applying f to such elements.

The application of this lemma is not usually found in the pencil and paper proofs of the "Rank nullity theorem", but will be crucial to know that, being f a linear form from a finite dimensional vector space V to a vector space V', and given a basis B of *ker f*, when B is completed up to a basis of V with a set W, the cardinal of this set is equal to the cardinal of its range set:

lemma *inj-on-extended*:
  assumes *lf*: *linear f*
  and *f*: *finite C*
  and *ind-C*: *independent C*
  and *C-eq*: $C = B \cup W$
  and *disj-set*: $B \cap W = \{\}$
  and *span-B*: $\{x.\ f\ x = 0\} \subseteq span\ B$
  shows *inj-on f W*
  — The proof is carried out by reductio ad absurdum
proof (*unfold inj-on-def*, *rule+*, *rule ccontr*)
  — Some previous consequences of the premises that are used later:
  have *fin-B*: *finite B* **using** *finite-subset* [*OF - f*] *C-eq* **by** *simp*
  have *ind-B*: *independent B* **and** *ind-W*: *independent W*
    using *independent-mono*[*OF ind-C*] *C-eq* **by** *simp-all*
  — The proof starts here; we assume that there exist two different elements
  — with the same image:
  fix *x*::*'a* **and** *y*::*'a*
  assume *x*: $x \in W$ **and** *y*: $y \in W$ **and** *f-eq*: $f\ x = f\ y$ **and** *x-not-y*: $x \neq y$
  have *fin-yB*: *finite* (*insert y B*) **using** *fin-B* **by** *simp*
  have $f\ (x - y) = 0$ **by** (*metis diff-self f-eq lf linear-0 linear-sub*)
  hence $x - y \in \{x.\ f\ x = 0\}$ **by** *simp*
  hence $\exists\, g.\ (\sum v{\in}B.\ g\ v *_R v) = (x - y)$ **using** *span-B*
    unfolding *span-finite* [*OF fin-B*] **by** *auto*
  then obtain *g* **where** *sum*: $(\sum v{\in}B.\ g\ v *_R v) = (x - y)$ **by** *blast*
  — We define one of the elements as a linear combination of the second element and the ones in B
  def $h \equiv (\lambda a.\ if\ a = y\ then\ (1::real)\ else\ g\ a)$
  have $x = y + (\sum v{\in}B.\ g\ v *_R v)$ **using** *sum* **by** *auto*
  also have ... $= h\ y *_R y\ + (\sum v{\in}B.\ g\ v *_R v)$ **unfolding** *h-def* **by** *simp*
  also have ... $= h\ y *_R y + (\sum v{\in}B.\ h\ v *_R v)$
    by (*unfold add-left-cancel*, *rule setsum-cong2*)
      (*metis* (*mono-tags*) *IntI disj-set empty-iff y h-def*)
  also have ... $= (\sum v{\in}(insert\ y\ B).\ h\ v *_R v)$
    by (*rule setsum-insert*[*symmetric*], *rule fin-B*)
      (*metis* (*lifting*) *IntI disj-set empty-iff y*)
  finally have *x-in-span-yB*: $x \in span\ (insert\ y\ B)$
    unfolding *span-finite*[*OF fin-yB*] **by** *auto*
  — We have that a subset of elements of C is linearly dependent
  have *dep*: *dependent* (*insert x* (*insert y B*))
    by (*unfold dependent-def*, *rule bexI* [*of - x*])
      (*metis Diff-insert-absorb Int-iff disj-set empty-iff insert-iff*
        x x-in-span-yB x-not-y, *simp*)
  — Therefore, the set C is also dependent:
  hence *dependent C* **using** *C-eq x y*
   by (*metis Un-commute Un-upper2 dependent-mono insert-absorb insert-subset*)
  — This yields the contradiction, since C is independent:
  thus *False* **using** *ind-C* **by** *contradiction*
qed

1.2 The proof

Now the rank nullity theorem can be proved; given any linear form *f*, the sum of the dimensions of its kernel and range subspaces is equal to the dimension of the source vector space.

It is relevant to note that the source vector space must be finite-dimensional (this restriction is introduced by means of the euclidean space type class), whereas the destination vector space may be finite or infinite dimensional (and thus a real vector space is used); this is the usual way the theorem is stated in the literature.

The statement of the "rank nullity theorem for linear algebra", as well as its proof, follow the ones on [1]. The proof is the traditional one found in the literature. The theorem is also named "fundamental theorem of linear algebra" in some texts (for instance, in [2]).

theorem *rank-nullity-theorem*:
  assumes *l*: *linear* (*f*::(*'a*::{*euclidean-space*}) => (*'b*::{*real-vector*}))
  shows *DIM* (*'a*::{*euclidean-space*}) = *dim* {*x*. *f x* = *0*} + *dim* (*range f*)
proof –
  — For convenience we define abbreviations for the universe set, V, and the kernel of f
  def *V* == *UNIV*::*'a set*
  def *ker-f* == {*x*. *f x* = *0*}
  — The kernel is a proper subspace:
  have *sub-ker*: *subspace* {*x*. *f x* = *0*} **using** *subspace-kernel* [*OF l*] **.**
  — The kernel has its proper basis, B:
  obtain *B* **where** *B-in-ker*: *B* ⊆ {*x*. *f x* = *0*}
    and *independent-B*: *independent B*
    and *ker-in-span*:{*x*. *f x* = *0*} ⊆ *span B*
    and *card-B*: *card B* = *dim* {*x*. *f x* = *0*} **using** *basis-exists* **by** *blast*
  — The space V has a (finite dimensional) basis, C:
  obtain *C* **where** *B-in-C*: *B* ⊆ *C* **and** *C-in-V*: *C* ⊆ *V*
    and *independent-C*: *independent C*
    and *span-C*: *V* = *span C*
    using *maximal-independent-subset-extend* [*OF - independent-B*, *of V*]
    unfolding *V-def* **by** *auto*
  — The basis of V, C, can be decomposed in the disjoint union of the basis of the kernel, B, and its complementary set, $C - B$
  have *C-eq*: *C* = *B* ∪ (*C* – *B*) **by** (*rule Diff-partition* [*OF B-in-C*, *symmetric*])
  have *eq-fC*: *f* ' *C* = *f* ' *B* ∪ *f* ' (*C* – *B*)
    by (*subst C-eq*, *unfold image-Un*, *simp*)
  — The basis C, and its image, are finite, since V is finite-dimensional
  have *finite-C*: *finite C*
    using *independent-bound-general* [*OF independent-C*] **by** *fast*
  have *finite-fC*: *finite* (*f* ' *C*) **by** (*rule finite-imageI* [*OF finite-C*])
  — The basis B of the kernel of f, and its image, are also finite
  have *finite-B*: *finite B* **by** (*rule rev-finite-subset* [*OF finite-C B-in-C*])

have *finite-fB*: *finite* (*f* ‘ *B*) **by** (*rule finite-imageI*[*OF finite-B*])
— The set $C - B$ is also finite
have *finite-CB*: *finite* (*C* − *B*) **by** (*rule finite-Diff* [*OF finite-C*, *of B*])
have *dim-ker-le-dim-V*:*dim* (*ker-f*) ≤ *dim V*
using *dim-subset* [*of ker-f V*] **unfolding** *V-def* **by** *simp*
— Here it starts the proof of the theorem: the sets B and $C - B$ must be proven to be bases, respectively, of the kernel of f and its range
show *?thesis*
proof −
have *DIM* ('*a*::{*euclidean-space*}) = *dim V* **unfolding** *V-def dim-UNIV* **..**
also have *dim V* = *dim C* **unfolding** *span-C dim-span* **..**
also have ... = *card C*
using *basis-card-eq-dim* [*of C C*, *OF - span-inc independent-C*] **by** *simp*
also have ... = *card* (*B* ∪ (*C* − *B*)) **using** *C-eq* **by** *simp*
also have ... = *card B* + *card* (*C*−*B*)
by (*rule card-Un-disjoint*[*OF finite-B finite-CB*], *fast*)
also have ... = *dim ker-f* + *card* (*C*−*B*) **unfolding** *ker-f-def card-B* **..**
— Now it has to be proved that the elements of $C - B$ are a basis of the range of f
also have ... = *dim ker-f* + *dim* (*range f*)
proof (*unfold add-left-cancel*)
def *W* == *C* − *B*
have *finite-W*: *finite W* **unfolding** *W-def* **using** *finite-CB* **.**
have *finite-fW*: *finite* (*f* ‘ *W*) **using** *finite-imageI*[*OF finite-W*] **.**
have *card W* = *card* (*f* ‘ *W*)
by (*rule card-image* [*symmetric*], *rule inj-on-extended* [*of - C B*],
rule l, *rule finite-C*)
(*rule independent-C*, *unfold W-def*, *subst C-eq*, *rule refl*, *simp*,
rule ker-in-span)
also have ... = *dim* (*range f*)
proof (*unfold dim-def*, *rule someI2*)
— 1. The image set of W generates the range of f:
have *range-in-span-fW*: *range f* ⊆ *span* (*f* ‘ *W*)
proof (*unfold span-finite* [*OF finite-fW*], *auto*)
— Given any element v in V, its image can be expressed as a linear combination of elements of the image by f of C:
fix *v* :: '*a*
have *fV-span*: *f* ‘ *V* ⊆ *span* (*f* ‘ *C*)
using *spans-image* [*OF l*] *span-C* **by** *simp*
have ∃ *g*. ($\sum$ *x*∈*f*‘*C*. *g x* $*_R$ *x*) = *f v*
using *fV-span* **unfolding** *V-def*
using *span-finite* [*OF finite-fC*] **by** *blast*
then obtain *g* **where** *fv*: *f v* = ($\sum$ *x*∈*f* ‘ *C*. *g x* $*_R$ *x*) **by** *metis*
— We recall that C is equal to B union $(C - B)$, and B is the basis of the kernel; thus, the image of the elements of B will be equal to zero:
have *zero-fB*: ($\sum$ *x*∈*f* ‘ *B*. *g x* $*_R$ *x*) = *0*
using *B-in-ker* **by** (*auto intro*!: *setsum-0*')
have *zero-inter*: ($\sum$ *x*∈(*f* ‘ *B* ∩ *f* ‘ *W*). *g x* $*_R$ *x*) = *0*
using *B-in-ker* **by** (*auto intro*!: *setsum-0*')

have $f\ v = (\sum x{\in}f\ `\ C.\ g\ x *_R x)$ **using** *fv* **.**
also have ... $= (\sum x{\in}(f\ `\ B \cup f\ `\ W).\ g\ x *_R x)$
using *eq-fC W-def* **by** *simp*
also have ... $=$
$(\sum x{\in}f\ `\ B.\ g\ x *_R x) + (\sum x{\in}f\ `\ W.\ g\ x *_R x) - (\sum x{\in}(f\ `\ B \cap f\ `\ W).\ g\ x *_R x)$
using *setsum-Un* [*OF finite-fB finite-fW*] **by** *simp*
also have ... $= (\sum x{\in}f\ `\ W.\ g\ x *_R x)$
unfolding *zero-fB zero-inter* **by** *simp*
— We have proved that the image set of W is a generating set of the range of f
finally show $\exists\, s.\ (\sum x{\in}f\ `\ W.\ s\ x *_R x) = f\ v$ **by** *auto*
qed
— 2. The image set of W is linearly independent:
have *independent-fW*: *independent* $(f\ `\ W)$
proof (*rule independent-if-scalars-zero* [*OF finite-fW*], *rule+*)
— Every linear combination (given by *gx*) of the elements of the image set of W equal to zero, requires every coefficient to be zero:
fix g :: $'b$ => *real* **and** w :: $'b$
assume *sum*: $(\sum x{\in}f\ `\ W.\ g\ x *_R x) = 0$ **and** $w$: $w \in f\ `\ W$
have $0 = (\sum x{\in}f\ `\ W.\ g\ x *_R x)$ **using** *sum* **by** *simp*
also have ... $=$ *setsum* $((\lambda x.\ g\ x *_R x) \circ f)\ W$
by (*rule setsum-reindex*, *rule inj-on-extended* [*of f C B*], *rule l*)
(*unfold W-def*, *rule finite-C*, *rule independent-C*, *rule C-eq*, *simp*, *rule ker-in-span*)
also have ... $= (\sum x{\in}W.\ ((g \circ f)\ x) *_R f\ x)$ **unfolding** *o-def* **..**
also have ... $= f\ (\sum x{\in}W.\ ((g \circ f)\ x) *_R x)$
using *linear-setsum-mul* [*symmetric*, *OF l finite-W*] **.**
finally have *f-sum-zero*:$f\ (\sum x{\in}W.\ (g \circ f)\ x *_R x) = 0$ **by** (*rule sym*)
hence $(\sum x{\in}W.\ (g \circ f)\ x *_R x) \in$ *ker-f* **unfolding** *ker-f-def* **by** *simp*
hence $\exists\, h.\ (\sum v{\in}B.\ h\ v *_R v) = (\sum x{\in}W.\ (g \circ f)\ x *_R x)$
using *span-finite*[*OF finite-B*] **using** *ker-in-span*
unfolding *ker-f-def* **by** *auto*
then obtain h **where**
sum-h: $(\sum v{\in}B.\ h\ v *_R v) = (\sum x{\in}W.\ (g \circ f)\ x *_R x)$ **by** *blast*
def $t \equiv (\lambda a.$ *if* $a \in B$ *then* $h\ a$ *else* $- ((g \circ f)\ a))$
have $0 = (\sum v{\in}B.\ h\ v *_R v) + - (\sum x{\in}W.\ (g \circ f)\ x *_R x)$
using *sum-h* **by** *simp*
also have ... $= (\sum v{\in}B.\ h\ v *_R v) + (\sum x{\in}W.\ -((g \circ f)\ x *_R x))$
unfolding *setsum-negf* **..**
also have ... $= (\sum v{\in}B.\ t\ v *_R v) + (\sum x{\in}W.\ -((g \circ f)\ x *_R x))$
unfolding *add-right-cancel* **unfolding** *t-def* **by** *simp*
also have ... $= (\sum v{\in}B.\ t\ v *_R v) + (\sum x{\in}W.\ t\ x *_R x)$
by (*unfold add-left-cancel t-def W-def*, *rule setsum-cong2*) *simp*
also have ... $= (\sum v{\in}B \cup W.\ t\ v *_R v)$
by (*rule setsum-Un-zero* [*symmetric*], *rule finite-B*, *rule finite-W*)
(*simp add*: *W-def*)
finally have $(\sum v{\in}B \cup W.\ t\ v *_R v) = 0$ **by** *simp*
hence *coef-zero*: $\forall\, x{\in}B \cup W.\ t\ x = 0$

```
      using C-eq scalars-zero-if-independent [OF finite-C independent-C]
      unfolding W-def by simp
    obtain y where w-fy: w = f y and y-in-W: y ∈ W using w by fast
    have − g w = t y
      unfolding t-def w-fy using y-in-W unfolding W-def by simp
    also have ... = 0 using coef-zero y-in-W unfolding W-def by simp
    finally show g w = 0 by simp
   qed
```

— The image set of W is independent and its span contains the range of f, so it is a basis of the range:

```
  show ∃ B⊆range f. independent B ∧ range f ⊆ span B
    ∧ card B = card (f ‘ W)
    by (rule exI [of -(f ‘ W)],
        simp add: range-in-span-fW independent-fW image-mono)
```

— Now, it has to be proved that any other basis of the subspace range of f has equal cardinality:

```
  show ⋀n::nat. ∃ S⊆range f. independent S ∧ range f ⊆ span S
    ∧ card S = n ⟹ card (f ‘ W) = n
  proof (clarify)
    fix S :: 'b set
    assume S-in-range: S ⊆ range f and independent-S: independent S
      and range-in-spanS: range f ⊆ span S
    have S-le: finite S ∧ card S ≤ card (f ‘ W)
    by (rule independent-span-bound, rule finite-fW, rule independent-S)
      (rule subset-trans [OF S-in-range range-in-span-fW])
    show card (f ‘ W) = card S
      by (rule le-antisym) (rule conjunct2, rule independent-span-bound,
        rule conjunct1 [OF S-le], rule independent-fW,
        rule subset-trans [OF - range-in-spanS], auto simp add: S-le)
  qed
 qed
 finally show card (C − B) = dim (range f) unfolding W-def .
qed
finally show ?thesis unfolding V-def ker-f-def unfolding dim-UNIV .
qed
qed
```

1.3 The rank nullity theorem for matrices

The previous lemma can be moved to the matrices' representation of linear forms; we introduce first the notions of null space (or kernel) and range (or column space) for matrices. The result

linear f $\Longrightarrow$ *card Basis* $=$ *dim* $\{x.\ f\ x = (0{::}'b)\}$ $+$ *dim* (*range f*)

is more general than its corresponding version for matrices representing linear forms, since in the first one the destination vector space could be finite or infinite dimensional, whereas in its version for matrices, both the source and destination vector spaces have to be finite-dimensional.

The null space correponds to the kernel of the linear form, and is a subset of the "row space":

definition *null-space* :: *real^'n^'m => (real^'n) set*
 where *null-space A* = {*x. A *v x = 0*}

The column space is a subset of the destination vector space of the linear form:

definition *col-space* :: *real^'n^'m=>(real^'m) set*
 where *col-space A = range* (λ*x. A *v x*)

lemma *col-space-eq*: *col-space A* = {*y*. ∃ *x. A *v x = y*}
 unfolding *col-space-def* **by** *blast*

lemma *null-space-eq-ker*:
 assumes *lf*: *linear f*
 shows *null-space* (*matrix f*) = {*x. f x = 0*}
 unfolding *null-space-def* **using** *matrix-works* [*OF lf*] **by** *auto*

lemma *col-space-eq-range*:
 assumes *lf*: *linear f*
 shows *col-space* (*matrix f*) = *range f*
 unfolding *col-space-def* **using** *matrix-works*[*OF lf*] **by** *auto*

After the previous equivalences between the null space and the column space and the range, the proof of the theorem for matrices is direct, as a consequence of the "rank nullity theorem".

lemma *rank-nullity-theorem-matrices*:
 fixes *A*::*real^'a^'b*
 shows *DIM* (*real^'a*) = *dim* (*null-space A*) + *dim* (*col-space A*)
 apply (*subst* (*1 2*) *matrix-of-matrix-vector-mul* [*of A*, *symmetric*])
 unfolding *null-space-eq-ker*[*OF matrix-vector-mul-linear*]
 unfolding *col-space-eq-range* [*OF matrix-vector-mul-linear*]
 using *rank-nullity-theorem* [*OF matrix-vector-mul-linear*] **.**

end

theory *Elementary-Operations*
 imports *Main*
 ~~/src/HOL/Multivariate-Analysis/Linear-Algebra
 ~~/src/HOL/Multivariate-Analysis/Cartesian-Euclidean-Space
 Dim-Formula
 Numeral-Type-Addenda
begin

2 Previous definitions

definition *nrows* :: *'a^'columns^'rows => nat*

where *nrows A* = *CARD*(*'rows*)

definition *ncols* :: *'a^'columns^'rows* => *nat*
where *ncols A* = *CARD*(*'columns*)

lemma *nrows-not-0*[*simp*]:
shows *0* ≠ *nrows A* **unfolding** *nrows-def* **by** *simp*

lemma *ncols-not-0*[*simp*]:
shows *0* ≠ *ncols A* **unfolding** *ncols-def* **by** *simp*

3 Code Generation for matrices

lemma [*code abstype*]: *vec-lambda* (*vec-nth v*) = (*v*::(*'a*::{*type*}, *'b*::{*finite*}) *vec*)
using *vec-lambda-eta* **by** *fast*

lemma [*code abstract*]: *vec-nth 0* = (%*x*. *0*) **by** (*metis zero-index*)
lemma [*code abstract*]: *vec-nth* (*a* + *b*) = (%*i*. *a*$*i* + *b*$*i*) **by** (*metis vector-add-component*)

definition *mat-mult-row*
where *mat-mult-row m m' f* = *vec-lambda* (%*c*. *setsum* (%*k*. ((*m*$*f*)$*k*) * ((*m'*$*k*)$*c*)) (*UNIV* :: *'n*::*finite set*))

lemma *mat-mult-row-code* [*code abstract*]:
vec-nth (*mat-mult-row m m' f*) = (%*c*. *setsum* (%*k*. ((*m*$*f*)$*k*) * ((*m'*$*k*)$*c*)) (*UNIV* :: *'n*::*finite set*))
by(*simp add*: *mat-mult-row-def fun-eq-iff*)

lemma *mat-mult* [*code abstract*]: *vec-nth* (*m* ** *m'*) = *mat-mult-row m m'*
unfolding *matrix-matrix-mult-def mat-mult-row-def*[*abs-def*]
using *vec-lambda-beta* **by** *auto*

lemma [*code abstract*]: *vec-nth* (*a* + *b*) = (%*i*. *a* $ *i* + *b* $ *i*) **by** (*metis vector-add-component*)

4 Elementary Operations

Some previous results:

lemma *invertible-mult*:
assumes *inv-A*: *invertible A*
and *inv-B*: *invertible B*
shows *invertible* (*A****B*)
proof −
obtain *A'* **where** *AA'*: *A* ** *A'* = *mat 1* **and** *A'A*: *A'* ** *A* = *mat 1* **using** *inv-A* **unfolding** *invertible-def* **by** *blast*
obtain *B'* **where** *BB'*: *B* ** *B'* = *mat 1* **and** *B'B*: *B'* ** *B* = *mat 1* **using** *inv-B* **unfolding** *invertible-def* **by** *blast*
show *?thesis*

proof (*unfold invertible-def*, *rule exI*[*of* - B'**A'], *rule conjI*)
have A ** B ** (B' ** A') = A ** (B ** (B' ** A')) **using** *matrix-mul-assoc*[*of* A B (B' ** A'), *symmetric*] **.**
also have ... = A ** (B ** B' ** A') **unfolding** *matrix-mul-assoc*[*of* B B' A'] **..**
also have ... = A ** (*mat 1* ** A') **unfolding** BB' **..**
also have ... = A ** A' **unfolding** *matrix-mul-lid* **..**
also have ... = *mat 1* **unfolding** AA' **..**
finally show A ** B ** (B' ** A') = *mat* (*1*::'*a*) **.**
have B' ** A' ** (A ** B) = B' ** (A' ** (A ** B)) **using** *matrix-mul-assoc*[*of* B' A' (A ** B), *symmetric*] **.**
also have ... = B' ** (A' ** A ** B) **unfolding** *matrix-mul-assoc*[*of* A' A B] **..**
also have ... = B' ** (*mat 1* ** B) **unfolding** $A'A$ **..**
also have ... = B' ** B **unfolding** *matrix-mul-lid* **..**
also have ... = *mat 1* **unfolding** $B'B$ **..**
finally show B' ** A' ** (A ** B) = *mat 1* **.**
qed
qed

In the library, *matrix-inv ?A* = (*SOME* A'. *?A* ** A' = *mat* (*1*::*?*'*a*) ∧ A' ** *?A* = *mat* (*1*::*?*'*a*)) allows the use of non squary matrices. The following lemma can be also proved fixing A

lemma *matrix-inv-unique*:
fixes *A*::'*a*::{*semiring-1*} ^'*n* ^'*n*
assumes *AB*: *A****B*=*mat 1* **and** *BA*: *B****A*=*mat 1*
shows *matrix-inv A* = *B*
proof (*unfold matrix-inv-def*, *rule some-equality*)
show A ** B = *mat* (*1*::'*a*) ∧ B ** A = *mat* (*1*::'*a*) **using** *AB BA* **by** *simp*
fix C **assume** A ** C = *mat* (*1*::'*a*) ∧ C ** A = *mat* (*1*::'*a*)
hence *AC*: A ** C = *mat* (*1*::'*a*) **and** *CA*: C ** A = *mat* (*1*::'*a*) **by** *auto*
have B = B ** (*mat 1*) **unfolding** *matrix-mul-rid* **..**
also have ... = B ** (A**C) **unfolding** *AC* **..**
also have ... = B ** A ** C **unfolding** *matrix-mul-assoc* **..**
also have ... = C **unfolding** *BA matrix-mul-lid* **..**
finally show C = B **..**
qed

lemma *mat-1-fun*: *mat 1* \$ *a* \$ *b* = (λ*i j*. *if i*=*j then 1 else 0*) *a b* **unfolding** *mat-def* **by** *auto*

lemma *mat1-sum-eq*:
shows (∑ *k*∈*UNIV*. *mat* (*1*::'*a*::{*semiring-1*}) \$ *s* \$ *k* ∗ *mat 1* \$ *k* \$ *t*) = *mat 1* \$ *s* \$ *t*
proof (*unfold mat-def*, *auto*)
let *?f*=λ*k*. (*if t* = *k then 1*::'*a else* (*0*::'*a*)) ∗ (*if k* = *t then 1*::'*a else* (*0*::'*a*))
have *univ-eq*: *UNIV* = (*UNIV* − {*t*}) ∪ {*t*} **by** *fast*
have *setsum ?f UNIV* = *setsum ?f* ((*UNIV* − {*t*}) ∪ {*t*}) **using** *univ-eq* **by**

simp
also have ... = *setsum ?f* (*UNIV* − {*t*}) + *setsum ?f* {*t*} **by** (*rule setsum-Un-disjoint*, *auto*)
also have ... = *0* + *setsum ?f* {*t*} **by** *auto*
also have ... = *setsum ?f* {*t*} **by** *simp*
also have ... = *1* **by** *simp*
finally show *setsum ?f UNIV* = *1* **.**
next
assume *s-not-t*: $s \neq t$
let *?g*=λk. (*if s* = *k then 1*::'*a else 0*) ∗ (*if k* = *t then 1 else 0*)
have *setsum ?g UNIV* = *setsum* (λk. *0*::'*a*) (*UNIV*::'*b set*) **by** (*rule setsum-cong2*, *simp add*: *s-not-t*)
also have ... = *0* **by** *simp*
finally show *setsum ?g UNIV* = *0* **.**
qed

lemma *finite-rows*: *finite* (*rows A*)
proof −
def *f*≡λi. *row i A*
show *?thesis* **unfolding** *rows-def* **using** *finite-Atleast-Atmost-nat*[*of f*] **unfolding** *f-def* **by** *simp*
qed

lemma *finite-columns*: *finite* (*columns A*)
proof −
def *f*≡λi. *column i A*
show *?thesis* **unfolding** *columns-def* **using** *finite-Atleast-Atmost-nat*[*of f*] **unfolding** *f-def* **by** *simp*
qed

lemma *invertible-mat-n*:
fixes *n*::'*a*::{*field*}
assumes *n*: $n \neq 0$
shows *invertible* ((*mat n*)::'*a*^'*n*^'*n*)
proof (*unfold invertible-def*, *rule exI*[*of - mat* (*inverse n*)], *rule conjI*)
show *mat n* ∗∗ *mat* (*inverse n*) = (*mat 1*::'*a*^'*n*^'*n*)
proof (*unfold matrix-matrix-mult-def mat-def*, *vector*, *auto*)
fix *ia*::'*n*
let *?f*=(λk. (*if ia* = *k then n else 0*) ∗ (*if k* = *ia then inverse n else 0*))
have *UNIV-rw*: (*UNIV*::'*n set*) = *insert ia* (*UNIV*−{*ia*}) **by** *auto*
have ($\sum k$∈(*UNIV*::'*n set*). (*if ia* = *k then n else 0*) ∗ (*if k* = *ia then inverse n else 0*)) =
($\sum k$∈*insert ia* (*UNIV*−{*ia*}). (*if ia* = *k then n else 0*) ∗ (*if k* = *ia then inverse n else 0*)) **using** *UNIV-rw* **by** *simp*
also have ... = *?f ia* + *setsum ?f* (*UNIV*−{*ia*})
proof (*rule setsum.insert*)
show *finite* (*UNIV* − {*ia*}) **using** *finite-UNIV* **by** *fastforce*

show *ia* $\notin$ *UNIV* $-$ $\{ia\}$ **by** *fast*
qed
also have ... = *1* **using** *right-inverse*[*OF n*] **by** *simp*
finally show ($\sum k\in$(*UNIV*::$'n$ *set*). (*if ia* = *k then n else 0*) $*$ (*if k* = *ia then inverse n else 0*)) = (*1*::$'a$) **.**
fix *i*::$'n$
assume *i-not-ia*: $i \neq ia$
show ($\sum k\in$(*UNIV*::$'n$ *set*). (*if i* = *k then n else 0*) $*$ (*if k* = *ia then inverse n else 0*)) = *0* **by** (*rule setsum-0'*, *simp add*: *i-not-ia*)
qed
next
show *mat* (*inverse n*) $**$ *mat n* = ((*mat 1*)::$'a$^$'n$^$'n$)
proof (*unfold matrix-matrix-mult-def mat-def*, *vector*, *auto*)
fix *ia*::$'n$
let *?f* = (λk. (*if ia* = *k then inverse n else 0*) $*$ (*if k* = *ia then n else 0*))
have *UNIV-rw*: (*UNIV*::$'n$ *set*) = *insert ia* (*UNIV*$-$$\{ia\}$) **by** *auto*
have ($\sum k\in$(*UNIV*::$'n$ *set*). (*if ia* = *k then inverse n else 0*) $*$ (*if k* = *ia then n else 0*)) =
($\sum k\in$*insert ia* (*UNIV*$-$$\{ia\}$). (*if ia* = *k then inverse n else 0*) $*$ (*if k* = *ia then n else 0*)) **using** *UNIV-rw* **by** *simp*
also have ... = *?f ia* + *setsum ?f* (*UNIV*$-$$\{ia\}$)
proof (*rule setsum.insert*)
show *finite* (*UNIV* $-$ $\{ia\}$) **using** *finite-UNIV* **by** *fastforce*
show *ia* $\notin$ *UNIV* $-$ $\{ia\}$ **by** *fast*
qed
also have ... = *1* **using** *left-inverse*[*OF n*] **by** *simp*
finally show ($\sum k\in$(*UNIV*::$'n$ *set*). (*if ia* = *k then inverse n else 0*) $*$ (*if k* = *ia then n else 0*)) = (*1*::$'a$) **.**
fix *i*::$'n$
assume *i-not-ia*: $i \neq ia$
show ($\sum k\in$(*UNIV*::$'n$ *set*). (*if i* = *k then inverse n else 0*) $*$ (*if k* = *ia then n else 0*)) = *0* **by** (*rule setsum-0'*, *simp add*: *i-not-ia*)
qed
qed

corollary *invertible-mat-1*:
shows *invertible* (*mat* (*1*::$'a$::{*field*})) **by** (*metis invertible-mat-n zero-neq-one*)

Functions between two real vector spaces form a real vector

instantiation *fun* :: (*real-vector*, *real-vector*) *real-vector*
begin

definition *plus-fun f g* = (λi. *f i* + *g i*)
definition *zero-fun* = (λi. *0*)
definition *scaleR-fun a f* = (λi. $a *_R f\ i$)

instance proof
fix *a*::$'a \Rightarrow 'b$ **and** *b*::$'a \Rightarrow 'b$ **and** *c*::$'a \Rightarrow 'b$
show *a* + *b* + *c* = *a* + (*b* + *c*) **unfolding** *fun-eq-iff* **unfolding** *plus-fun-def*

by *auto*
show $a + b = b + a$ **unfolding** *fun-eq-iff* **unfolding** *plus-fun-def* **by** *auto*
show $(0::'a \Rightarrow 'b) + a = a$ **unfolding** *fun-eq-iff* **unfolding** *plus-fun-def zero-fun-def* **by** *auto*
show $- a + a = (0::'a \Rightarrow 'b)$ **unfolding** *fun-eq-iff* **unfolding** *plus-fun-def zero-fun-def* **by** *auto*
show $a - b = a + - b$ **unfolding** *fun-eq-iff* **unfolding** *plus-fun-def zero-fun-def* **by** *auto*
next
fix a::*real* **and** x::$('a \Rightarrow 'b)$ **and** y::$'a \Rightarrow 'b$
show $a *_R (x + y) = a *_R x + a *_R y$
unfolding *fun-eq-iff plus-fun-def scaleR-fun-def scaleR-right.add* **by** *auto*
next
fix a::*real* **and** b::*real* **and** x::$'a \Rightarrow 'b$
show $(a + b) *_R x = a *_R x + b *_R x$
unfolding *fun-eq-iff* **unfolding** *plus-fun-def scaleR-fun-def* **unfolding** *scaleR-left.add* **by** *auto*
show $a *_R b *_R x = (a * b) *_R x$ **unfolding** *fun-eq-iff* **unfolding** *scaleR-fun-def* **by** *auto*
show $(1::real) *_R x = x$ **unfolding** *fun-eq-iff* **unfolding** *scaleR-fun-def* **by** *auto*
qed
end

Definitions of elementary row operations

definition *interchange-rows* :: ('a::*semiring-1*) ^'n^'m => 'm => 'm ⇒ 'a ^'n^'m
where *interchange-rows A a b* = (χ *i j. if i=a then A \$ b \$ j else if i=b then A \$ a \$ j else A \$ i \$ j*)

definition *mult-row* :: ('a::*semiring-1*) ^'n^'m => 'm => 'a ⇒ 'a ^'n^'m
where *mult-row A a q* = (χ *i j. if i=a then q*∗(*A \$ a \$ j*) *else A \$ i \$ j*)

definition *row-add* :: ('a::*semiring-1*) ^'n^'m => 'm => 'm ⇒ 'a ⇒ 'a ^'n^'m
where *row-add A a b q* = (χ *i j. if i=a then* (*A \$ a \$ j*) + *q*∗(*A \$ b \$ j*) *else A \$ i \$ j*)

Definitions of elementary column operations

definition *interchange-columns* :: ('a::*semiring-1*) ^'n^'m => 'n => 'n ⇒ 'a ^'n^'m
where *interchange-columns A n m* = (χ *i j. if j=n then A \$ i \$ m else if j=m then A \$ i \$ n else A \$ i \$ j*)

definition *mult-column* :: ('a::*semiring-1*) ^'n^'m => 'n => 'a ⇒ 'a ^'n^'m
where *mult-column A n q* = (χ *i j. if j=n then* (*A \$ i \$ j*)∗*q else A \$ i \$ j*)

definition *column-add* :: ('a::*semiring-1*) ^'n^'m => 'n => 'n ⇒ 'a ⇒ 'a ^'n^'m
where *column-add A n m q* = (χ *i j. if j=n then* ((*A \$ i \$ n*) + (*A \$ i \$ m*)∗*q*) *else A \$ i \$ j*)

Properties about *interchange-rows*

lemma *interchange-same-rows*: *interchange-rows A a a = A*
  unfolding *interchange-rows-def* **by** *vector*

lemma *interchange-rows-i*[*simp*]: *interchange-rows A i j* $ *i = A* $ *j*
  unfolding *interchange-rows-def* **by** *vector*

lemma *interchange-rows-j*[*simp*]: *interchange-rows A i j* $ *j = A* $ *i*
  unfolding *interchange-rows-def* **by** *vector*

lemma *interchange-rows-preserves*:
  assumes $i \neq a$ **and** $j \neq a$
  shows *interchange-rows A i j* $ *a = A* $ *a*
  using *assms* **unfolding** *interchange-rows-def* **by** *vector*

lemma *interchange-rows-mat-1*:
  shows *interchange-rows* (*mat 1*) *a b* ** *A = interchange-rows A a b*
proof (*unfold matrix-matrix-mult-def interchange-rows-def*, *vector*, *auto*)
  fix *ia*
  let *?f*=(λ*k*. *mat* (*1*::'*a*) $ *a* $ *k* ∗ *A* $ *k* $ *ia*)
  have *univ-rw*:*UNIV* = (*UNIV*−{*a*}) ∪ {*a*} **by** *auto*
  have *setsum ?f UNIV* = *setsum ?f* ((*UNIV*−{*a*}) ∪ {*a*}) **using** *univ-rw* **by** *auto*
  also have ... = *setsum ?f* (*UNIV*−{*a*}) + *setsum ?f* {*a*}
  proof (*rule setsum-Un-disjoint*)
    show *finite* (*UNIV* − {*a*}) **by** (*metis finite-code*)
    show *finite* {*a*} **by** *simp*
    show (*UNIV* − {*a*}) ∩ {*a*} = {} **by** *simp*
  qed
  also have ... = *setsum ?f* {*a*} **unfolding** *mat-def* **by** *auto*
  also have ... = *?f a* **by** *auto*
  also have ... = *A* $ *a* $ *ia* **unfolding** *mat-def* **by** *auto*
  finally show (∑*k*∈*UNIV*. *mat* (*1*::'*a*) $ *a* $ *k* ∗ *A* $ *k* $ *ia*) = *A* $ *a* $ *ia* **.**
  assume *i*: $a \neq b$
  let *?g*= λ*k*. *mat* (*1*::'*a*) $ *b* $ *k* ∗ *A* $ *k* $ *ia*
  have *univ-rw'*:*UNIV* = (*UNIV*−{*b*}) ∪ {*b*} **by** *auto*
  have *setsum ?g UNIV* = *setsum ?g* ((*UNIV*−{*b*}) ∪ {*b*}) **using** *univ-rw'* **by** *auto*
  also have ... = *setsum ?g* (*UNIV*−{*b*}) + *setsum ?g* {*b*} **by** (*rule setsum-Un-disjoint*, *auto*)
  also have ... = *setsum ?g* {*b*} **unfolding** *mat-def* **by** *auto*
  also have ... = *?g b* **by** *simp*
  finally show (∑*k*∈*UNIV*. *mat* (*1*::'*a*) $ *b* $ *k* ∗ *A* $ *k* $ *ia*) = *A* $ *b* $ *ia* **unfolding** *mat-def* **by** *simp*
next
  fix *i j*
  assume *ib*: $i \neq b$ **and** *ia*:$i \neq a$
  let *?h*=λ*k*. *mat* (*1*::'*a*) $ *i* $ *k* ∗ *A* $ *k* $ *j*
  have *univ-rw''*:*UNIV* = (*UNIV*−{*i*}) ∪ {*i*} **by** *auto*
  have *setsum ?h UNIV* = *setsum ?h* ((*UNIV*−{*i*}) ∪ {*i*}) **using** *univ-rw''* **by**

auto
also have ... = *setsum ?h* (*UNIV*−{*i*}) + *setsum ?h* {*i*} **by** (*rule setsum-Un-disjoint*, *auto*)
also have ... = *setsum ?h* {*i*} **unfolding** *mat-def* **by** *auto*
also have ... = *?h i* **by** *simp*
finally show ($\sum$ *k*∈*UNIV*. *mat* (*1*::′*a*) $ *i* $ *k* ∗ *A* $ *k* $ *j*) = *A* $ *i* $ *j* **unfolding** *mat-def* **by** *auto*
qed

lemma *invertible-interchange-rows*: *invertible* (*interchange-rows* (*mat 1*) *a b*)
proof (*unfold invertible-def*, *rule exI*[*of - interchange-rows* (*mat 1*) *a b*], *simp*, *unfold matrix-matrix-mult-def*, *vector*, *clarify*,
unfold interchange-rows-def, *vector*, *unfold mat-1-fun*, *auto*+)
fix *s t*::′*b*
assume *s-not-t*: *s* ≠ *t*
show ($\sum$ *k*::′*b*∈*UNIV*. (*if s* = *k then 1*::′*a else* (*0*::′*a*)) ∗ (*if k* = *t then 1*::′*a else if k* = *t then 1*::′*a else if k* = *t then 1*::′*a else* (*0*::′*a*))) = (*0*::′*a*)
by (*rule setsum-0*′, *simp add*: *s-not-t*)
assume *b-not-t*: *b* ≠ *t*
show ($\sum$ *k*∈*UNIV*. (*if s* = *b then if t* = *k then 1*::′*a else* (*0*::′*a*) *else if s* = *k then 1*::′*a else* (*0*::′*a*)) ∗
(*if k* = *t then 0*::′*a else if k* = *b then 1*::′*a else if k* = *t then 1*::′*a else* (*0*::′*a*))) =
(*0*::′*a*) **by** (*rule setsum-0*′, *simp*)
assume *a-not-t*: *a* ≠ *t*
show ($\sum$ *k*∈*UNIV*. (*if s* = *a then if b* = *k then 1*::′*a else* (*0*::′*a*) *else if s* = *b then if a* = *k then 1*::′*a else* (*0*::′*a*) *else if s* = *k then 1*::′*a else* (*0*::′*a*)) ∗
(*if k* = *a then 0*::′*a else if k* = *b then 0*::′*a else if k* = *t then 1*::′*a else* (*0*::′*a*))) =
(*0*::′*a*) **by** (*rule setsum-0*′, *auto simp add*: *s-not-t*)
next
fix *s t*::′*b*
assume *a-noteq-t*: *a*≠*t* **and** *s-noteq-t*: *s* ≠ *t*
show ($\sum$ *k*∈*UNIV*. (*if s* = *a then if t* = *k then 1*::′*a else* (*0*::′*a*) *else if s* = *t then if a* = *k then 1*::′*a else* (*0*::′*a*) *else if s* = *k then 1*::′*a else* (*0*::′*a*)) ∗
(*if k* = *a then 1*::′*a else if k* = *t then 0*::′*a else if k* = *t then 1*::′*a else* (*0*::′*a*))) =
(*0*::′*a*) **apply** (*rule setsum-0*′) **using** *s-noteq-t* **by** *fastforce*
next
fix *s t*::′*b*
show ($\sum$ *k*∈*UNIV*. (*if t* = *k then 1*::′*a else* (*0*::′*a*)) ∗ (*if k* = *t then 1*::′*a else if k* = *t then 1*::′*a else if k* = *t then 1*::′*a else* (*0*::′*a*))) = (*1*::′*a*)
proof −
let *?f*=(λ*k*. (*if t* = *k then 1*::′*a else* (*0*::′*a*)) ∗ (*if k* = *t then 1*::′*a else if k* = *t then 1*::′*a else if k* = *t then 1*::′*a else* (*0*::′*a*)))
have *univ-eq*: *UNIV* = ((*UNIV* − {*t*}) ∪ {*t*}) **by** *auto*
have *setsum ?f UNIV* = *setsum ?f* ((*UNIV* − {*t*}) ∪ {*t*}) **using** *univ-eq* **by** *simp*
also have ... = *setsum ?f* (*UNIV* − {*t*}) + *setsum ?f* {*t*} **by** (*rule setsum-Un-disjoint*,

auto)
  also have ... = *0* + *setsum ?f* {*t*} **by** *auto*
  also have ... = *setsum ?f* {*t*} **by** *simp*
  also have ... = *1* **by** *simp*
  finally show *?thesis* **.**
 qed
next
 fix *s t*::'*b*
 assume *b-noteq-t*: $b \neq t$
 show $(\sum k\in UNIV.\ (if\ b = k\ then\ 1::'a\ else\ (0::'a)) * (if\ k = t\ then\ 0::'a\ else\ if\ k = b\ then\ 1::'a\ else\ if\ k = t\ then\ 1::'a\ else\ (0::'a))) = (1::'a)$
 proof −
  let $?f=(\lambda k.\ (if\ b = k\ then\ 1::'a\ else\ (0::'a)) * (if\ k = t\ then\ 0::'a\ else\ if\ k = b\ then\ 1::'a\ else\ if\ k = t\ then\ 1::'a\ else\ (0::'a)))$
  have *univ-eq*: $UNIV = ((UNIV - \{b\}) \cup \{b\})$ **by** *auto*
  have $setsum\ ?f\ UNIV = setsum\ ?f\ ((UNIV - \{b\}) \cup \{b\})$ **using** *univ-eq* **by** *simp*
  also have ... = $setsum\ ?f\ (UNIV - \{b\}) + setsum\ ?f\ \{b\}$ **by** (*rule setsum-Un-disjoint*, *auto*)
  also have ... = *0* + *setsum ?f* {*b*} **by** *auto*
  also have ... = *setsum ?f* {*b*} **by** *simp*
  also have ... = *1* **using** *b-noteq-t* **by** *simp*
  finally show *?thesis* **.**
 qed
 assume *a-noteq-t*: $a\neq t$
 show $(\sum k\in UNIV.\ (if\ t = k\ then\ 1::'a\ else\ (0::'a)) * (if\ k = a\ then\ 0::'a\ else\ if\ k = b\ then\ 0::'a\ else\ if\ k = t\ then\ 1::'a\ else\ (0::'a))) = (1::'a)$
 proof −
  let $?f=(\lambda k.\ (if\ t = k\ then\ 1::'a\ else\ (0::'a)) * (if\ k = a\ then\ 0::'a\ else\ if\ k = b\ then\ 0::'a\ else\ if\ k = t\ then\ 1::'a\ else\ (0::'a)))$
  have *univ-eq*: $UNIV = ((UNIV - \{t\}) \cup \{t\})$ **by** *auto*
  have $setsum\ ?f\ UNIV = setsum\ ?f\ ((UNIV - \{t\}) \cup \{t\})$ **using** *univ-eq* **by** *simp*
  also have ... = $setsum\ ?f\ (UNIV - \{t\}) + setsum\ ?f\ \{t\}$ **by** (*rule setsum-Un-disjoint*, *auto*)
  also have ... = *0* + *setsum ?f* {*t*} **by** *auto*
  also have ... = *setsum ?f* {*t*} **by** *simp*
  also have ... = *1* **using** *b-noteq-t a-noteq-t* **by** *simp*
  finally show *?thesis* **.**
 qed
next
 fix *s t*::'*b*
 assume *a-noteq-t*: $a\neq t$
 show $(\sum k\in UNIV.\ (if\ a = k\ then\ 1::'a\ else\ (0::'a)) * (if\ k = a\ then\ 1::'a\ else\ if\ k = t\ then\ 0::'a\ else\ if\ k = t\ then\ 1::'a\ else\ (0::'a))) = (1::'a)$
 proof −
  let $?f=\lambda k.\ (if\ a = k\ then\ 1::'a\ else\ (0::'a)) * (if\ k = a\ then\ 1::'a\ else\ if\ k = t\ then\ 0::'a\ else\ if\ k = t\ then\ 1::'a\ else\ (0::'a))$
  have *univ-eq*: $UNIV = ((UNIV - \{a\}) \cup \{a\})$ **by** *auto*

have *setsum ?f UNIV* = *setsum ?f* ((*UNIV* − {*a*}) ∪ {*a*}) **using** *univ-eq* **by** *simp*
also have ... = *setsum ?f* (*UNIV* − {*a*}) + *setsum ?f* {*a*} **by** (*rule setsum-Un-disjoint*, *auto*)
also have ... = *0* + *setsum ?f* {*a*} **by** *auto*
also have ... = *setsum ?f* {*a*} **by** *simp*
also have ... = *1* **using** *a-noteq-t* **by** *simp*
finally show *?thesis* **.**
qed
qed

Properties about *mult-row*

lemma *mult-row-mat-1*: *mult-row* (*mat 1*) *a q* ∗∗ *A* = *mult-row A a q*
proof (*unfold matrix-matrix-mult-def mult-row-def*, *vector*, *auto*)
fix *ia*
let *?f*=λ*k*. *q* ∗ *mat* (*1*::′*a*) $ *a* $ *k* ∗ *A* $ *k* $ *ia*
have *univ-rw*:*UNIV* = (*UNIV*−{*a*}) ∪ {*a*} **by** *auto*
have *setsum ?f UNIV* = *setsum ?f* ((*UNIV*−{*a*}) ∪ {*a*}) **using** *univ-rw* **by** *auto*
also have ... = *setsum ?f* (*UNIV*−{*a*}) + *setsum ?f* {*a*} **by** (*rule setsum-Un-disjoint*, *auto*)
also have ... = *setsum ?f* {*a*} **unfolding** *mat-def* **by** *auto*
also have ... = *?f a* **by** *auto*
also have ... = *q* ∗ *A* $ *a* $ *ia* **unfolding** *mat-def* **by** *auto*
finally show ($\sum$ *k*∈*UNIV*. *q* ∗ *mat* (*1*::′*a*) $ *a* $ *k* ∗ *A* $ *k* $ *ia*) = *q* ∗ *A* $ *a* $ *ia* **.**
fix *i*
assume *i*: *i* ≠ *a*
let *?g*=λ*k*. *mat* (*1*::′*a*) $ *i* $ *k* ∗ *A* $ *k* $ *ia*
have *univ-rw''*:*UNIV* = (*UNIV*−{*i*}) ∪ {*i*} **by** *auto*
have *setsum ?g UNIV* = *setsum ?g* ((*UNIV*−{*i*}) ∪ {*i*}) **using** *univ-rw''* **by** *auto*
also have ... = *setsum ?g* (*UNIV*−{*i*}) + *setsum ?g* {*i*} **by** (*rule setsum-Un-disjoint*, *auto*)
also have ... = *setsum ?g* {*i*} **unfolding** *mat-def* **by** *auto*
also have ... = *?g i* **by** *simp*
finally show ($\sum$ *k*∈*UNIV*. *mat* (*1*::′*a*) $ *i* $ *k* ∗ *A* $ *k* $ *ia*) = *A* $ *i* $ *ia* **unfolding** *mat-def* **by** *simp*
qed

lemma *invertible-mult-row*:
assumes *qk*: *q*∗*k*=*1* **and** *kq*: *k*∗*q*=*1*
shows *invertible* (*mult-row* (*mat 1*) *a q*)
proof (*unfold invertible-def*, *rule exI*[*of - mult-row* (*mat 1*) *a k*],*rule conjI*)
show *mult-row* (*mat* (*1*::′*a*)) *a q* ∗∗ *mult-row* (*mat* (*1*::′*a*)) *a k* = *mat* (*1*::′*a*)
proof (*unfold matrix-matrix-mult-def*, *vector*, *clarify*, *unfold mult-row-def*, *vector*, *unfold mat-1-fun*, *auto*)
show ($\sum$ *ka*∈*UNIV*. *q* ∗ (*if a* = *ka then 1*::′*a else* (*0*::′*a*)) ∗ (*if ka* = *a then k* ∗ (*1*::′*a*) *else if ka* = *a then 1*::′*a else* (*0*::′*a*)))) = (*1*::′*a*)

proof −
let *?f*=λ*ka*. *q* ∗ (*if a* = *ka then 1*::′*a else* (*0*::′*a*)) ∗ (*if ka* = *a then k* ∗ (*1*::′*a*) *else if ka* = *a then 1*::′*a else* (*0*::′*a*))
have *univ-eq*: *UNIV* = ((*UNIV* − {*a*}) ∪ {*a*}) **by** *auto*
have *setsum ?f UNIV* = *setsum ?f* ((*UNIV* − {*a*}) ∪ {*a*}) **using** *univ-eq* **by** *simp*
also have ... = *setsum ?f* (*UNIV* − {*a*}) + *setsum ?f* {*a*} **by** (*rule setsum-Un-disjoint*, *auto*)
also have ... = *0* + *setsum ?f* {*a*} **by** *auto*
also have ... = *setsum ?f* {*a*} **by** *simp*
also have ... = *1* **using** *qk* **by** *simp*
finally show *?thesis* **.**
qed
next
fix *s*
assume *s-noteq-a*: *s*≠*a*
show ($\sum$*ka*∈*UNIV*. (*if s* = *ka then 1*::′*a else* (*0*::′*a*)) ∗ (*if ka* = *a then k* ∗ (*1*::′*a*) *else if ka* = *a then 1*::′*a else 0*)) = *0*
by (*rule setsum-0*′, *simp add*: *s-noteq-a*)
next
fix *t*
assume *a-noteq-t*: *a*≠*t*
show ($\sum$*ka*∈*UNIV*. (*if t* = *ka then 1*::′*a else* (*0*::′*a*)) ∗ (*if ka* = *a then k* ∗ (*0*::′*a*) *else if ka* = *t then 1*::′*a else* (*0*::′*a*))) = (*1*::′*a*)
proof −
let *?f*=λ*ka*. (*if t* = *ka then 1*::′*a else* (*0*::′*a*)) ∗ (*if ka* = *a then k* ∗ (*0*::′*a*) *else if ka* = *t then 1*::′*a else* (*0*::′*a*))
have *univ-eq*: *UNIV* = ((*UNIV* − {*t*}) ∪ {*t*}) **by** *auto*
have *setsum ?f UNIV* = *setsum ?f* ((*UNIV* − {*t*}) ∪ {*t*}) **using** *univ-eq* **by** *simp*
also have ... = *setsum ?f* (*UNIV* − {*t*}) + *setsum ?f* {*t*} **by** (*rule setsum-Un-disjoint*, *auto*)
also have ... = *setsum ?f* {*t*} **by** *simp*
also have ... = *1* **using** *a-noteq-t* **by** *auto*
finally show *?thesis* **.**
qed
fix *s*
assume *s-not-t*: *s*≠*t*
show ($\sum$*ka*∈*UNIV*. (*if s* = *a then q* ∗ (*if a* = *ka then 1*::′*a else* (*0*::′*a*)) *else if s* = *ka then 1*::′*a else* (*0*::′*a*)) ∗
(*if ka* = *a then k* ∗ (*0*::′*a*) *else if ka* = *t then 1*::′*a else* (*0*::′*a*))) = (*0*::′*a*)
by (*rule setsum-0*′, *simp add*: *s-not-t a-noteq-t*)
qed
show *mult-row* (*mat* (*1*::′*a*)) *a k* ∗∗ *mult-row* (*mat* (*1*::′*a*)) *a q* = *mat* (*1*::′*a*)
proof (*unfold matrix-matrix-mult-def*, *vector*, *clarify*, *unfold mult-row-def*, *vector*, *unfold mat-1-fun*, *auto*)
show ($\sum$*ka*∈*UNIV*. *k* ∗ (*if a* = *ka then 1*::′*a else* (*0*::′*a*)) ∗ (*if ka* = *a then q* ∗ (*1*::′*a*) *else if ka* = *a then 1*::′*a else* (*0*::′*a*))) = (*1*::′*a*)
proof −

let *?f*=λ*ka*. *k* ∗ (*if a* = *ka then 1*::′*a else* (*0*::′*a*)) ∗ (*if ka* = *a then q* ∗ (*1*::′*a*) *else if ka* = *a then 1*::′*a else* (*0*::′*a*))
have *univ-eq*: *UNIV* = ((*UNIV* − {*a*}) ∪ {*a*}) **by** *auto*
have *setsum ?f UNIV* = *setsum ?f* ((*UNIV* − {*a*}) ∪ {*a*}) **using** *univ-eq* **by** *simp*
also have ... = *setsum ?f* (*UNIV* − {*a*}) + *setsum ?f* {*a*} **by** (*rule setsum-Un-disjoint*, *auto*)
also have ... = *0* + *setsum ?f* {*a*} **by** *auto*
also have ... = *setsum ?f* {*a*} **by** *simp*
also have ... = *1* **using** *kq* **by** *simp*
finally show *?thesis* **.**
qed
next
fix *s*
assume *s-not-a*: *s*≠*a*
show (∑ *k*∈*UNIV*. (*if s* = *k then 1*::′*a else* (*0*::′*a*)) ∗ (*if k* = *a then q* ∗ (*1*::′*a*) *else if k* = *a then 1*::′*a else* (*0*::′*a*))) = (*0*::′*a*)
by (*rule setsum-0*′, *simp add*: *s-not-a*)
next
fix *t*
assume *a-not-t*: *a*≠*t*
show (∑ *k*∈*UNIV*. (*if t* = *k then 1*::′*a else* (*0*::′*a*)) ∗ (*if k* = *a then q* ∗ (*0*::′*a*) *else if k* = *t then 1*::′*a else* (*0*::′*a*))) = (*1*::′*a*)
proof −
let *?f*=λ*k*. (*if t* = *k then 1*::′*a else* (*0*::′*a*)) ∗ (*if k* = *a then q* ∗ (*0*::′*a*) *else if k* = *t then 1*::′*a else* (*0*::′*a*))
have *univ-eq*: *UNIV* = ((*UNIV* − {*t*}) ∪ {*t*}) **by** *auto*
have *setsum ?f UNIV* = *setsum ?f* ((*UNIV* − {*t*}) ∪ {*t*}) **using** *univ-eq* **by** *simp*
also have ... = *setsum ?f* (*UNIV* − {*t*}) + *setsum ?f* {*t*} **by** (*rule setsum-Un-disjoint*, *auto*)
also have ... = *setsum ?f* {*t*} **by** *simp*
also have ... = *1* **using** *a-not-t* **by** *simp*
finally show *?thesis* **.**
qed
fix *s*
assume *s-not-t*: *s*≠*t*
show (∑ *ka*∈*UNIV*. (*if s* = *a then k* ∗ (*if a* = *ka then 1*::′*a else* (*0*::′*a*)) *else if s* = *ka then 1*::′*a else* (*0*::′*a*)) ∗
(*if ka* = *a then q* ∗ (*0*::′*a*) *else if ka* = *t then 1*::′*a else* (*0*::′*a*))) = (*0*::′*a*)
by (*rule setsum-0*′, *simp add*: *s-not-t*)
qed
qed

corollary *invertible-mult-row*′:
assumes *q-not-zero*: *q* ≠ *0*
shows *invertible* (*mult-row* (*mat* (*1*::′*a*::{*field*})) *a q*)
by (*simp add*: *invertible-mult-row*[*of q inverse q*] *q-not-zero*)

Properties about *row-add*

lemma *row-add-mat-1*: *row-add* (*mat 1*) *a b q* ∗∗ *A* = *row-add A a b q*
proof (*unfold matrix-matrix-mult-def row-add-def*, *vector*, *auto*)
  fix *j*
  let *?f*= (λ*k*. (*mat* (*1*::′*a*) $ *a* $ *k* + *q* ∗ *mat* (*1*::′*a*) $ *b* $ *k*) ∗ *A* $ *k* $ *j*)
  show *setsum ?f UNIV* = *A* $ *a* $ *j* + *q* ∗ *A* $ *b* $ *j*
  proof (*cases a*=*b*)
    case *False*
    have *univ-rw*: *UNIV* = {*a*} ∪ ({*b*} ∪ (*UNIV* − {*a*} − {*b*})) **by** *auto*
    have *setsum-rw*: *setsum ?f* ({*b*} ∪ (*UNIV* − {*a*} − {*b*})) = *setsum ?f* {*b*} + *setsum ?f* (*UNIV* − {*a*} − {*b*}) **by** (*rule setsum-Un-disjoint*, *auto simp add*: *False*)
    have *setsum ?f UNIV* = *setsum ?f* ({*a*} ∪ ({*b*} ∪ (*UNIV* − {*a*} − {*b*}))) **using** *univ-rw* **by** *simp*
    also have ... = *setsum ?f* {*a*} + *setsum ?f* ({*b*} ∪ (*UNIV* − {*a*} − {*b*})) **by** (*rule setsum-Un-disjoint*, *auto simp add*: *False*)
    also have ... = *setsum ?f* {*a*} + *setsum ?f* {*b*} + *setsum ?f* (*UNIV* − {*a*} − {*b*}) **unfolding** *setsum-rw ab-semigroup-add-class.add-ac*(*1*)[*symmetric*] **..**
    also have ... = *setsum ?f* {*a*} + *setsum ?f* {*b*}
    proof −
      have *setsum ?f* (*UNIV* − {*a*} − {*b*}) = *setsum* (λ*k*. *0*) (*UNIV* − {*a*} − {*b*}) **unfolding** *mat-def* **by** (*rule setsum-cong2*, *auto*)
      also have ... = *0* **unfolding** *setsum-0* **..**
      finally show *?thesis* **by** *simp*
    qed
    also have ... = *A* $ *a* $ *j* + *q* ∗ *A* $ *b* $ *j* **using** *False* **unfolding** *mat-def* **by** *simp*
    finally show *?thesis* **.**
  next
    case *True*
    have *univ-rw*: *UNIV* = {*b*} ∪ (*UNIV* − {*b*}) **by** *auto*
    have *setsum ?f UNIV* = *setsum ?f* ({*b*} ∪ (*UNIV* − {*b*})) **using** *univ-rw* **by** *simp*
    also have ... = *setsum ?f* {*b*} + *setsum ?f* (*UNIV* − {*b*}) **by** (*rule setsum-Un-disjoint*, *auto*)
    also have ... = *setsum ?f* {*b*}
    proof −
      have *setsum ?f* (*UNIV* − {*b*}) = *setsum* (λ*k*. *0*) (*UNIV* − {*b*}) **using** *True* **unfolding** *mat-def* **by** *auto*
      also have ... = *0* **unfolding** *setsum-0* **..**
      finally show *?thesis* **by** *simp*
    qed
    also have ... = *A* $ *a* $ *j* + *q* ∗ *A* $ *b* $ *j*
    by (*unfold True mat-def*, *simp*, *metis* (*hide-lams*, *no-types*) *vector-add-component vector-sadd-rdistrib vector-smult-component vector-smult-lid*)
    finally show *?thesis* **.**
  qed
  fix *i* **assume** *i*: *i*≠*a*
  let *?g*=λ*k*. *mat* (*1*::′*a*) $ *i* $ *k* ∗ *A* $ *k* $ *j*
  have *univ-rw*: *UNIV* = {*i*} ∪ (*UNIV* − {*i*}) **by** *auto*

have *setsum ?g UNIV = setsum ?g* ({*i*} ∪ (*UNIV* − {*i*})) **using** *univ-rw* **by** *simp*
also have ... = *setsum ?g* {*i*} + *setsum ?g* (*UNIV* − {*i*}) **by** (*rule setsum-Un-disjoint*, *auto*)
also have ... = *setsum ?g* {*i*}
proof −
have *setsum ?g* (*UNIV* − {*i*}) = *setsum* (λ*k*. *0*) (*UNIV* − {*i*}) **unfolding** *mat-def* **by** *auto*
also have ... = *0* **unfolding** *setsum-0* **..**
finally show *?thesis* **by** *simp*
qed
also have ... = *A* $ *i* $ *j* **unfolding** *mat-def* **by** *simp*
finally show ($\sum$*k*∈*UNIV*. *mat* (*1*::′*a*) $ *i* $ *k* ∗ *A* $ *k* $ *j*) = *A* $ *i* $ *j* **.**
qed

lemma *invertible-row-add*:
assumes *a-noteq-b*: *a*≠*b*
shows *invertible* (*row-add* (*mat* (*1*::′*a*::{*ring-1*})) *a b q*)
proof (*unfold invertible-def*, *rule exI*[*of* - (*row-add* (*mat 1*) *a b* (−*q*))], *rule conjI*)
show *row-add* (*mat* (*1*::′*a*)) *a b q* ∗∗ *row-add* (*mat* (*1*::′*a*)) *a b* (− *q*) = *mat* (*1*::′*a*) **using** *a-noteq-b*
proof (*unfold matrix-matrix-mult-def*, *vector*, *clarify*, *unfold row-add-def*, *vector*, *unfold mat-1-fun*, *auto*)
show ($\sum$*k*::′*b*∈*UNIV*. (*if b* = *k then 1*::′*a else* (*0*::′*a*)) ∗ (*if k* = *a then* (*0*::′*a*) + − *q* ∗ (*1*::′*a*) *else if k* = *b then 1*::′*a else* (*0*::′*a*))) = (*1*::′*a*)
proof −
let *?f*=λ*k*. (*if b* = *k then 1*::′*a else* (*0*::′*a*)) ∗ (*if k* = *a then* (*0*::′*a*) + − *q* ∗ (*1*::′*a*) *else if k* = *b then 1*::′*a else* (*0*::′*a*))
have *univ-eq*: *UNIV* = ((*UNIV* − {*b*}) ∪ {*b*}) **by** *auto*
have *setsum ?f UNIV* = *setsum ?f* ((*UNIV* − {*b*}) ∪ {*b*}) **using** *univ-eq* **by** *simp*
also have ... = *setsum ?f* (*UNIV* − {*b*}) + *setsum ?f* {*b*} **by** (*rule setsum-Un-disjoint*, *auto*)
also have ... = *0* + *setsum ?f* {*b*} **by** *auto*
also have ... = *setsum ?f* {*b*} **by** *simp*
also have ... = *1* **using** *a-noteq-b* **by** *simp*
finally show *?thesis* **.**
qed
show ($\sum$*k*::′*b*∈*UNIV*. ((*if a* = *k then 1*::′*a else* (*0*::′*a*)) + *q* ∗ (*if b* = *k then 1*::′*a else* (*0*::′*a*))) ∗ (*if k* = *a then* (*1*::′*a*) + −
q ∗ (*0*::′*a*) *else if k* = *a then 1*::′*a else* (*0*::′*a*))) = (*1*::′*a*)
proof −
let *?f*=λ*k*. ((*if a* = *k then 1*::′*a else* (*0*::′*a*)) + *q* ∗ (*if b* = *k then 1*::′*a else* (*0*::′*a*))) ∗ (*if k* = *a then* (*1*::′*a*) + −
q ∗ (*0*::′*a*) *else if k* = *a then 1*::′*a else* (*0*::′*a*))
have *univ-eq*: *UNIV* = ((*UNIV* − {*a*}) ∪ {*a*}) **by** *auto*
have *setsum ?f UNIV* = *setsum ?f* ((*UNIV* − {*a*}) ∪ {*a*}) **using** *univ-eq* **by** *simp*
also have ... = *setsum ?f* (*UNIV* − {*a*}) + *setsum ?f* {*a*} **by** (*rule*

setsum-Un-disjoint, *auto*)
  also have ... = *0* + *setsum ?f* {*a*} **by** *auto*
  also have ... = *setsum ?f* {*a*} **by** *simp*
  also have ... = *1* **using** *a-noteq-b* **by** *simp*
  finally show *?thesis* **.**
 qed
next
 fix *s*
 assume *s-not-a*: $s \neq a$
 show ($\sum k::'b\in UNIV$. (*if* $s = k$ *then* $1::'a$ *else* $(0::'a)$) $*$ (*if* $k = a$ *then* $(1::'a)$ $+ - q * (0::'a)$ *else if* $k = a$ *then* $1::'a$ *else* $(0::'a)$))) $= (0::'a)$
  by (*rule setsum-0'*, *auto simp add*: *s-not-a*)
next
 fix *t*
 assume *b-not-t*: $b \neq t$ **and** *a-not-t*: $a \neq t$
 show ($\sum k\in UNIV$. (*if* $t = k$ *then* $1::'a$ *else* $(0::'a)$) $*$ (*if* $k = a$ *then* $(0::'a) +$ $- q * (0::'a)$ *else if* $k = t$ *then* $1::'a$ *else* $(0::'a)$))) $= (1::'a)$
 proof −
  let *?f*=λk. (*if* $t = k$ *then* $1::'a$ *else* $(0::'a)$) $*$ (*if* $k = a$ *then* $(0::'a) + - q *$ $(0::'a)$ *else if* $k = t$ *then* $1::'a$ *else* $(0::'a)$)
  have *univ-eq*: $UNIV = ((UNIV - \{t\}) \cup \{t\})$ **by** *auto*
  have *setsum ?f UNIV* = *setsum ?f* $((UNIV - \{t\}) \cup \{t\})$ **using** *univ-eq* **by** *simp*
  also have ... = *setsum ?f* $(UNIV - \{t\})$ + *setsum ?f* $\{t\}$ **by** (*rule setsum-Un-disjoint*, *auto*)
  also have ... = *0* + *setsum ?f* {*t*} **by** *auto*
  also have ... = *setsum ?f* {*t*} **by** *simp*
  also have ... = *1* **using** *b-not-t a-not-t* **by** *simp*
  finally show *?thesis* **.**
 qed
next
 fix *s t*
 assume *b-not-t*: $b \neq t$ **and** *a-not-t*: $a \neq t$ **and** *s-not-t*: $s \neq t$
 show ($\sum k\in UNIV$. (*if* $s = a$ *then* (*if* $a = k$ *then* $1::'a$ *else* $(0::'a)$) $+ q *$ (*if* $b = k$ *then* $1::'a$ *else* $(0::'a)$) *else if* $s = k$ *then* $1::'a$ *else* $(0::'a)$) $*$
  (*if* $k = a$ *then* $(0::'a) + - q * (0::'a)$ *else if* $k = t$ *then* $1::'a$ *else* $(0::'a)$))) $=$ $(0::'a)$ **by** (*rule setsum-0'*, *auto simp add*: *b-not-t a-not-t s-not-t*)
next
 fix *s*
 assume *s-not-b*: $s \neq b$
 let *?f*=λk. (*if* $s = a$ *then* (*if* $a = k$ *then* $1::'a$ *else* $(0::'a)$) $+ q *$ (*if* $b = k$ *then* $1::'a$ *else* $(0::'a)$) *else if* $s = k$ *then* $1::'a$ *else* $(0::'a)$) $*$
  (*if* $k = a$ *then* $(0::'a) + - q * (1::'a)$ *else if* $k = b$ *then* $1::'a$ *else* $(0::'a)$)
 show *setsum ?f UNIV* = $(0::'a)$
 proof (*cases* $s=a$)
  case *False*
  show *?thesis* **by** (*rule setsum-0'*, *auto simp add*: *False s-not-b a-noteq-b*)
 next
  case *True* — This case is different from the other cases

have *univ-eq*: $UNIV = ((UNIV - \{a\}- \{b\}) \cup (\{b\} \cup \{a\}))$ **by** *auto*
have *setsum-a*: *setsum ?f* $\{a\} = -q$ **unfolding** *True* **using** *s-not-b* **using** *a-noteq-b* **by** *auto*
have *setsum-b*: *setsum ?f* $\{b\} = q$ **unfolding** *True* **using** *s-not-b* **using** *a-noteq-b* **by** *auto*
have *setsum-rest*: *setsum ?f* $(UNIV - \{a\} - \{b\}) = 0$ **by** (*rule setsum-0'*, *auto simp add*: *True s-not-b a-noteq-b*)
have *setsum ?f UNIV* = *setsum ?f* $((UNIV - \{a\}- \{b\}) \cup (\{b\} \cup \{a\}))$ **using** *univ-eq* **by** *simp*
also have ... = *setsum ?f* $(UNIV - \{a\} - \{b\})$ + *setsum ?f* $(\{b\} \cup \{a\})$ **by** (*rule setsum-Un-disjoint*, *auto*)
also have ... = *setsum ?f* $(UNIV - \{a\} - \{b\})$ + *setsum ?f* $\{b\}$ + *setsum ?f* $\{a\}$ **by** (*auto simp add*: *setsum-Un-disjoint a-noteq-b*)
also have ... = *0* **unfolding** *setsum-a setsum-b setsum-rest* **by** *simp*
finally show *?thesis* **.**
qed
qed
next
show *row-add* (*mat* (*1*::'*a*)) *a b* (− *q*) ∗∗ *row-add* (*mat* (*1*::'*a*)) *a b q* = *mat* (*1*::'*a*) **using** *a-noteq-b*
proof (*unfold matrix-matrix-mult-def*, *vector*, *clarify*, *unfold row-add-def*, *vector*, *unfold mat-1-fun*, *auto*)
show ($\sum$ *k*∈*UNIV*. (*if b = k then 1*::'*a else* (*0*::'*a*)) ∗ (*if k = a then* (*0*::'*a*) + *q* ∗ (*1*::'*a*) *else if k = b then 1*::'*a else* (*0*::'*a*))) = (*1*::'*a*)
proof −
let *?f*=λ*k*. (*if b = k then 1*::'*a else* (*0*::'*a*)) ∗ (*if k = a then* (*0*::'*a*) + *q* ∗ (*1*::'*a*) *else if k = b then 1*::'*a else* (*0*::'*a*))
have *univ-eq*: $UNIV = ((UNIV - \{b\}) \cup \{b\})$ **by** *auto*
have *setsum ?f UNIV* = *setsum ?f* $((UNIV - \{b\}) \cup \{b\})$ **using** *univ-eq* **by** *simp*
also have ... = *setsum ?f* $(UNIV - \{b\})$ + *setsum ?f* $\{b\}$ **by** (*rule setsum-Un-disjoint*, *auto*)
also have ... = *0* + *setsum ?f* $\{b\}$ **by** *auto*
also have ... = *setsum ?f* $\{b\}$ **by** *simp*
also have ... = *1* **using** *a-noteq-b* **by** *simp*
finally show *?thesis* **.**
qed
next
show ($\sum$ *k*∈*UNIV*. ((*if a = k then 1*::'*a else* (*0*::'*a*)) + − (*q* ∗ (*if b = k then 1*::'*a else* (*0*::'*a*)))) ∗ (*if k = a then* (*1*::'*a*)
+ *q* ∗ (*0*::'*a*) *else if k = a then 1*::'*a else* (*0*::'*a*))) = (*1*::'*a*)
proof −
let *?f*=λ*k*. ((*if a = k then 1*::'*a else* (*0*::'*a*)) + − (*q* ∗ (*if b = k then 1*::'*a else* (*0*::'*a*)))) ∗ (*if k = a then* (*1*::'*a*) + *q* ∗ (*0*::'*a*) *else if k = a then 1*::'*a else* (*0*::'*a*))
have *univ-eq*: $UNIV = ((UNIV - \{a\}) \cup \{a\})$ **by** *auto*
have *setsum ?f UNIV* = *setsum ?f* $((UNIV - \{a\}) \cup \{a\})$ **using** *univ-eq* **by** *simp*
also have ... = *setsum ?f* $(UNIV - \{a\})$ + *setsum ?f* $\{a\}$ **by** (*rule*

setsum-Un-disjoint, *auto*)
also have ... = *0* + *setsum ?f* {*a*} **by** *auto*
also have ... = *setsum ?f* {*a*} **by** *simp*
also have ... = *1* **using** *a-noteq-b* **by** *simp*
finally show *?thesis* **.**
qed
next
fix *s*
assume *s-not-a*: $s \neq a$
show ($\sum k \in UNIV$. (*if* $s = k$ *then* $1::'a$ *else* $(0::'a)$) $*$ (*if* $k = a$ *then* $(1::'a) + q * (0::'a)$ *else if* $k = a$ *then* $1::'a$ *else* $(0::'a)$)) $= (0::'a)$
by (*rule setsum-0'*, *auto simp add*: *s-not-a*)
next
fix *t*
assume *b-not-t*: $b \neq t$ **and** *a-not-t*: $a \neq t$
show ($\sum k \in UNIV$. (*if* $t = k$ *then* $1::'a$ *else* $(0::'a)$) $*$ (*if* $k = a$ *then* $(0::'a) + q * (0::'a)$ *else if* $k = t$ *then* $1::'a$ *else* $(0::'a)$)) $= (1::'a)$
proof −
let $?f = \lambda k$. (*if* $t = k$ *then* $1::'a$ *else* $(0::'a)$) $*$ (*if* $k = a$ *then* $(0::'a) + q * (0::'a)$ *else if* $k = t$ *then* $1::'a$ *else* $(0::'a)$)
have *univ-eq*: $UNIV = ((UNIV - \{t\}) \cup \{t\})$ **by** *auto*
have *setsum ?f UNIV* = *setsum ?f* $((UNIV - \{t\}) \cup \{t\})$ **using** *univ-eq* **by** *simp*
also have ... = *setsum ?f* $(UNIV - \{t\})$ + *setsum ?f* $\{t\}$ **by** (*rule setsum-Un-disjoint*, *auto*)
also have ... = *0* + *setsum ?f* {*t*} **by** *auto*
also have ... = *setsum ?f* {*t*} **by** *simp*
also have ... = *1* **using** *b-not-t a-not-t* **by** *simp*
finally show *?thesis* **.**
qed
next
fix *s t*
assume *b-not-t*: $b \neq t$ **and** *a-not-t*: $a \neq t$ **and** *s-not-t*: $s \neq t$
show ($\sum k \in UNIV$. (*if* $s = a$ *then* (*if* $a = k$ *then* $1::'a$ *else* $(0::'a)$) $+ - q *$ (*if* $b = k$ *then* $1::'a$ *else* $(0::'a)$) *else if* $s = k$ *then* $1::'a$ *else* $(0::'a)$) $*$
(*if* $k = a$ *then* $(0::'a) + q * (0::'a)$ *else if* $k = t$ *then* $1::'a$ *else* $(0::'a)$)) $= (0::'a)$
by (*rule setsum-0'*, *auto simp add*: *b-not-t a-not-t s-not-t*)
next
fix *s*
assume *s-not-b*: $s \neq b$
let $?f = \lambda k$.(*if* $s = a$ *then* (*if* $a = k$ *then* $1::'a$ *else* $(0::'a)$) $+ - q *$ (*if* $b = k$ *then* $1::'a$ *else* $(0::'a)$) *else if* $s = k$ *then* $1::'a$ *else* $(0::'a)$)
$*$ (*if* $k = a$ *then* $(0::'a) + q * (1::'a)$ *else if* $k = b$ *then* $1::'a$ *else* $(0::'a)$)
show *setsum ?f UNIV* = *0*
proof (*cases* $s = a$)
case *False*
show *?thesis* **by** (*rule setsum-0'*, *auto simp add*: *False s-not-b a-noteq-b*)
next

case *True* — This case is different from the other cases
have *univ-eq*: $UNIV = ((UNIV - \{a\} - \{b\}) \cup (\{b\} \cup \{a\}))$ **by** *auto*
have *setsum-a*: *setsum ?f* $\{a\} = q$ **unfolding** *True* **using** *s-not-b* **using** *a-noteq-b* **by** *auto*
have *setsum-b*: *setsum ?f* $\{b\} = -q$ **unfolding** *True* **using** *s-not-b* **using** *a-noteq-b* **by** *auto*
have *setsum-rest*: *setsum ?f* $(UNIV - \{a\} - \{b\}) = 0$ **by** (*rule setsum-0'*, *auto simp add*: *True s-not-b a-noteq-b*)
have *setsum ?f UNIV* = *setsum ?f* $((UNIV - \{a\} - \{b\}) \cup (\{b\} \cup \{a\}))$ **using** *univ-eq* **by** *simp*
also have ... = *setsum ?f* $(UNIV - \{a\} - \{b\})$ + *setsum ?f* $(\{b\} \cup \{a\})$ **by** (*rule setsum-Un-disjoint*, *auto*)
also have ... = *setsum ?f* $(UNIV - \{a\} - \{b\})$ + *setsum ?f* $\{b\}$ + *setsum ?f* $\{a\}$ **by** (*auto simp add*: *setsum-Un-disjoint a-noteq-b*)
also have ... = 0 **unfolding** *setsum-a setsum-b setsum-rest* **by** *simp*
finally show *?thesis* **.**
qed
qed
qed

Properties about *interchange-columns*

lemma *interchange-columns-mat-1*: A ** *interchange-columns* (*mat 1*) *a b* = *interchange-columns A a b*
proof (*unfold matrix-matrix-mult-def*, *unfold interchange-columns-def*, *vector*, *auto*)

fix *i*
show $(\sum k \in UNIV.\ A \$ i \$ k * mat\ (1::'a) \$ k \$ a) = A \$ i \$ a$
proof −
let $?f = (\lambda k.\ A \$ i \$ k * mat\ (1::'a) \$ k \$ a)$
have *univ-rw*: $UNIV = (UNIV - \{a\}) \cup \{a\}$ **by** *auto*
have *setsum ?f UNIV* = *setsum ?f* $((UNIV - \{a\}) \cup \{a\})$ **using** *univ-rw* **by** *auto*
also have ... = *setsum ?f* $(UNIV - \{a\})$ + *setsum ?f* $\{a\}$ **by** (*rule setsum-Un-disjoint*, *auto*)
also have ... = *setsum ?f* $\{a\}$ **unfolding** *mat-def* **by** *auto*
finally show *?thesis* **unfolding** *mat-def* **by** *simp*
qed
assume *a-not-b*: $a \neq b$
show $(\sum k \in UNIV.\ A \$ i \$ k * mat\ (1::'a) \$ k \$ b) = A \$ i \$ b$
proof −
let $?f = (\lambda k.\ A \$ i \$ k * mat\ (1::'a) \$ k \$ b)$
have *univ-rw*: $UNIV = (UNIV - \{b\}) \cup \{b\}$ **by** *auto*
have *setsum ?f UNIV* = *setsum ?f* $((UNIV - \{b\}) \cup \{b\})$ **using** *univ-rw* **by** *auto*
also have ... = *setsum ?f* $(UNIV - \{b\})$ + *setsum ?f* $\{b\}$ **by** (*rule setsum-Un-disjoint*, *auto*)
also have ... = *setsum ?f* $\{b\}$ **unfolding** *mat-def* **by** *auto*
finally show *?thesis* **unfolding** *mat-def* **by** *simp*
qed

next
fix *i j*
assume *j-not-b*: *j* ≠ *b* **and** *j-not-a*: *j* ≠ *a*
show (∑ *k*∈*UNIV*. *A* $ *i* $ *k* ∗ *mat* (*1*::′*a*) $ *k* $ *j*) = *A* $ *i* $ *j*
proof −
let *?f*=(λ*k*. *A* $ *i* $ *k* ∗ *mat* (*1*::′*a*) $ *k* $ *j*)
have *univ-rw*:*UNIV* = (*UNIV*−{*j*}) ∪ {*j*} **by** *auto*
have *setsum ?f UNIV* = *setsum ?f* ((*UNIV*−{*j*}) ∪ {*j*}) **using** *univ-rw* **by** *auto*
also have ... = *setsum ?f* (*UNIV*−{*j*}) + *setsum ?f* {*j*} **by** (*rule setsum-Un-disjoint*, *auto*)
also have ... = *setsum ?f* {*j*} **unfolding** *mat-def* **using** *j-not-b j-not-a* **by** *auto*
finally show *?thesis* **unfolding** *mat-def* **by** *simp*
qed
qed

lemma *invertible-interchange-columns*: *invertible* (*interchange-columns* (*mat 1*) *a* *b*)
proof (*unfold invertible-def*, *rule exI*[*of - interchange-columns* (*mat 1*) *a b*], *simp*, *unfold matrix-matrix-mult-def*, *vector*, *clarify*,
unfold interchange-columns-def, *vector*, *unfold mat-1-fun*, *auto*+)
show (∑ *k*∈*UNIV*. (*if k* = *b then 1*::′*a else if k* = *b then 1*::′*a else if b* = *k then 1*::′*a else* (*0*::′*a*)) ∗ (*if k* = *b then 1*::′*a else* (*0*::′*a*))) = (*1*::′*a*)
proof −
let *?f*=(λ*k*. (*if k* = *b then 1*::′*a else if k* = *b then 1*::′*a else if b* = *k then 1*::′*a else* (*0*::′*a*)) ∗ (*if k* = *b then 1*::′*a else* (*0*::′*a*)))
have *univ-rw*:*UNIV* = (*UNIV*−{*b*}) ∪ {*b*} **by** *auto*
have *setsum ?f UNIV* = *setsum ?f* ((*UNIV*−{*b*}) ∪ {*b*}) **using** *univ-rw* **by** *auto*
also have ... = *setsum ?f* (*UNIV*−{*b*}) + *setsum ?f* {*b*} **by** (*rule setsum-Un-disjoint*, *auto*)
also have ... = *setsum ?f* {*b*} **by** *auto*
finally show *?thesis* **by** *simp*
qed
assume *a-not-b*: *a* ≠ *b*
show (∑ *k*∈*UNIV*. (*if k* = *a then 0*::′*a else if k* = *b then 1*::′*a else if a* = *k then 1*::′*a else* (*0*::′*a*)) ∗ (*if k* = *b then 1*::′*a else* (*0*::′*a*))) = (*1*::′*a*)
proof −
let *?f*=λ*k*. (*if k* = *a then 0*::′*a else if k* = *b then 1*::′*a else if a* = *k then 1*::′*a else* (*0*::′*a*)) ∗ (*if k* = *b then 1*::′*a else* (*0*::′*a*))
have *univ-rw*:*UNIV* = (*UNIV*−{*b*}) ∪ {*b*} **by** *auto*
have *setsum ?f UNIV* = *setsum ?f* ((*UNIV*−{*b*}) ∪ {*b*}) **using** *univ-rw* **by** *auto*
also have ... = *setsum ?f* (*UNIV*−{*b*}) + *setsum ?f* {*b*} **by** (*rule setsum-Un-disjoint*, *auto*)
also have ... = *setsum ?f* {*b*} **using** *a-not-b* **by** *simp*
finally show *?thesis* **using** *a-not-b* **by** *auto*
qed

next
fix *t*
assume *b-not-t*: *b* ≠ *t*
show (∑*k*∈*UNIV*. (*if k* = *b then 1*::'*a else if k* = *b then 1*::'*a else if b* = *k then 1*::'*a else* (*0*::'*a*)) ∗ (*if k* = *t then 1*::'*a else* (*0*::'*a*))) = (*0*::'*a*)
apply (*rule setsum-0'*) **using** *b-not-t* **by** *auto*
assume *b-not-a*: *b* ≠ *a*
show (∑*k*∈*UNIV*. (*if k* = *a then 1*::'*a else if k* = *b then 0*::'*a else if b* = *k then 1*::'*a else* (*0*::'*a*)) ∗
(*if t* = *a then if k* = *b then 1*::'*a else* (*0*::'*a*) *else if t* = *b then if k* = *a then 1*::'*a else* (*0*::'*a*) *else if k* = *t then 1*::'*a else* (*0*::'*a*))) =
(*0*::'*a*) **apply** (*rule setsum-0'*) **using** *b-not-t* **by** *auto*
next
fix *t*
assume *a-not-b*: *a* ≠ *b* **and** *a-not-t*: *a* ≠ *t*
show (∑*k*∈*UNIV*. (*if k* = *a then 0*::'*a else if k* = *b then 1*::'*a else if a* = *k then 1*::'*a else* (*0*::'*a*)) ∗
(*if t* = *b then if k* = *a then 1*::'*a else* (*0*::'*a*) *else if k* = *t then 1*::'*a else* (*0*::'*a*)))
= (*0*::'*a*)
by (*rule setsum-0'*, *auto simp add*: *a-not-b a-not-t*)
next
assume *b-not-a*: *b* ≠ *a*
show (∑*k*∈*UNIV*. (*if k* = *a then 1*::'*a else if k* = *b then 0*::'*a else if b* = *k then 1*::'*a else* (*0*::'*a*)) ∗ (*if k* = *a then 1*::'*a else* (*0*::'*a*))) = (*1*::'*a*)
proof −
let *?f*=λ*k*. (*if k* = *a then 1*::'*a else if k* = *b then 0*::'*a else if b* = *k then 1*::'*a else* (*0*::'*a*)) ∗ (*if k* = *a then 1*::'*a else* (*0*::'*a*))
have *univ-rw*:*UNIV* = (*UNIV*−{*a*}) ∪ {*a*} **by** *auto*
have *setsum ?f UNIV* = *setsum ?f* ((*UNIV*−{*a*}) ∪ {*a*}) **using** *univ-rw* **by** *auto*
also have ... = *setsum ?f* (*UNIV*−{*a*}) + *setsum ?f* {*a*} **by** (*rule setsum-Un-disjoint*, *auto*)
also have ... = *setsum ?f* {*a*} **using** *b-not-a* **by** *simp*
finally show *?thesis* **using** *b-not-a* **by** *auto*
qed
next
fix *t*
assume *t-not-a*: *t* ≠ *a* **and** *t-not-b*: *t* ≠ *b*
show (∑*k*∈*UNIV*. (*if k* = *a then 0*::'*a else if k* = *b then 0*::'*a else if t* = *k then 1*::'*a else* (*0*::'*a*)) ∗ (*if k* = *t then 1*::'*a else* (*0*::'*a*))) = (*1*::'*a*)
proof −
let *?f*=λ*k*. (*if k* = *a then 0*::'*a else if k* = *b then 0*::'*a else if t* = *k then 1*::'*a else* (*0*::'*a*)) ∗ (*if k* = *t then 1*::'*a else* (*0*::'*a*))
have *univ-rw*:*UNIV* = (*UNIV*−{*t*}) ∪ {*t*} **by** *auto*
have *setsum ?f UNIV* = *setsum ?f* ((*UNIV*−{*t*}) ∪ {*t*}) **using** *univ-rw* **by** *auto*
also have ... = *setsum ?f* (*UNIV*−{*t*}) + *setsum ?f* {*t*} **by** (*rule setsum-Un-disjoint*, *auto*)
also have ... = *setsum ?f* {*t*} **using** *t-not-a t-not-b* **by** *simp*

also have ... = *1* **using** *t-not-a t-not-b* **by** *simp*
finally show *?thesis* **.**
qed
next
fix *s t*
assume *s-not-a*: $s \neq a$ **and** *s-not-b*: $s \neq b$ **and** *s-not-t*: $s \neq t$
show ($\sum k \in UNIV$. (*if* $k = a$ *then* $0::'a$ *else if* $k = b$ *then* $0::'a$ *else if* $s = k$ *then* $1::'a$ *else* $(0::'a)$) $*$
(*if* $t = a$ *then if* $k = b$ *then* $1::'a$ *else* $(0::'a)$ *else if* $t = b$ *then if* $k = a$ *then* $1::'a$ *else* $(0::'a)$ *else if* $k = t$ *then* $1::'a$ *else* $(0::'a)$)) =
$(0::'a)$
by (*rule setsum-0'*, *auto simp add*: *s-not-a s-not-b s-not-t*)
qed

Properties about *mult-column*

lemma *mult-column-mat-1*: $A ** $ *mult-column* (*mat 1*) *a q* = *mult-column A a q*
proof (*unfold matrix-matrix-mult-def*, *unfold mult-column-def*, *vector*, *auto*)
fix *i*
show ($\sum k \in UNIV$. *A* \$ *i* \$ *k* $*$ (*mat* ($1::'a$) \$ *k* \$ *a* $*$ *q*)) = *A* \$ *i* \$ *a* $*$ *q*
proof −
let *?f*=λk. *A* \$ *i* \$ *k* $*$ (*mat* ($1::'a$) \$ *k* \$ *a* $*$ *q*)
have *univ-rw*:*UNIV* = (*UNIV*−{*a*}) ∪ {*a*} **by** *auto*
have *setsum ?f UNIV* = *setsum ?f* ((*UNIV*−{*a*}) ∪ {*a*}) **using** *univ-rw* **by** *auto*
also have ... = *setsum ?f* (*UNIV*−{*a*}) + *setsum ?f* {*a*} **by** (*rule setsum-Un-disjoint*, *auto*)
also have ... = *setsum ?f* {*a*} **unfolding** *mat-def* **by** *auto*
also have ... = *A* \$ *i* \$ *a* $*$ *q* **unfolding** *mat-def* **by** *auto*
finally show *?thesis* **.**
qed
fix *j*
show ($\sum k \in UNIV$. *A* \$ *i* \$ *k* $*$ *mat* ($1::'a$) \$ *k* \$ *j*) = *A* \$ *i* \$ *j*
proof −
let *?f*=λk. *A* \$ *i* \$ *k* $*$ *mat* ($1::'a$) \$ *k* \$ *j*
have *univ-rw*:*UNIV* = (*UNIV*−{*j*}) ∪ {*j*} **by** *auto*
have *setsum ?f UNIV* = *setsum ?f* ((*UNIV*−{*j*}) ∪ {*j*}) **using** *univ-rw* **by** *auto*
also have ... = *setsum ?f* (*UNIV*−{*j*}) + *setsum ?f* {*j*} **by** (*rule setsum-Un-disjoint*, *auto*)
also have ... = *setsum ?f* {*j*} **unfolding** *mat-def* **by** *auto*
also have ... = *A* \$ *i* \$ *j* **unfolding** *mat-def* **by** *auto*
finally show *?thesis* **.**
qed
qed

lemma *invertible-mult-column*:
assumes *qk*: *q*$*$*k*=*1* **and** *kq*: *k*$*$*q*=*1*
shows *invertible* (*mult-column* (*mat 1*) *a q*)
proof (*unfold invertible-def*, *rule exI*[*of - mult-column* (*mat 1*) *a k*], *rule conjI*)

show *mult-column* (*mat 1*) *a q* ∗∗ *mult-column* (*mat 1*) *a k* = *mat 1*
proof (*unfold matrix-matrix-mult-def*, *vector*, *clarify*, *unfold mult-column-def*, *vector*, *unfold mat-1-fun*, *auto*)
fix *t*
show ($\sum$ *ka*∈*UNIV*. (*if ka* = *a then* (*if t* = *ka then 1*::′*a else* (*0*::′*a*)) ∗ *q else if t* = *ka then 1*::′*a else* (*0*::′*a*)) ∗
(*if t* = *a then* (*if ka* = *t then 1*::′*a else* (*0*::′*a*)) ∗ *k else if ka* = *t then 1*::′*a else* (*0*::′*a*))) =
(*1*::′*a*)
proof −
let *?f*= λ*ka*. (*if ka* = *a then* (*if t* = *ka then 1*::′*a else* (*0*::′*a*)) ∗ *q else if t* = *ka then 1*::′*a else* (*0*::′*a*)) ∗
(*if t* = *a then* (*if ka* = *t then 1*::′*a else* (*0*::′*a*)) ∗ *k else if ka* = *t then 1*::′*a else* (*0*::′*a*))
have *univ-rw*:*UNIV* = (*UNIV*−{*t*}) ∪ {*t*} **by** *auto*
have *setsum ?f UNIV* = *setsum ?f* ((*UNIV*−{*t*}) ∪ {*t*}) **using** *univ-rw* **by** *auto*
also have ... = *setsum ?f* (*UNIV*−{*t*}) + *setsum ?f* {*t*} **by** (*rule setsum-Un-disjoint*, *auto*)
also have ... = *setsum ?f* {*t*} **by** *auto*
also have ... = *1* **using** *qk* **by** *auto*
finally show *?thesis* **.**
qed
fix *s*
assume *s-not-t*: *s* ≠ *t*
show ($\sum$ *ka*∈*UNIV*. (*if ka* = *a then* (*if s* = *ka then 1*::′*a else* (*0*::′*a*)) ∗ *q else if s* = *ka then 1*::′*a else* (*0*::′*a*)) ∗
(*if t* = *a then* (*if ka* = *t then 1*::′*a else* (*0*::′*a*)) ∗ *k else if ka* = *t then 1*::′*a else* (*0*::′*a*))) =
(*0*::′*a*)
apply (*rule setsum-0′*) **using** *s-not-t* **by** *auto*
qed
show *mult-column* (*mat* (*1*::′*a*)) *a k* ∗∗ *mult-column* (*mat* (*1*::′*a*)) *a q* = *mat* (*1*::′*a*)
proof (*unfold matrix-matrix-mult-def*, *vector*, *clarify*, *unfold mult-column-def*, *vector*, *unfold mat-1-fun*, *auto*)
fix *t*
show ($\sum$ *ka*∈*UNIV*. (*if ka* = *a then* (*if t* = *ka then 1*::′*a else* (*0*::′*a*)) ∗ *k else if t* = *ka then 1*::′*a else* (*0*::′*a*)) ∗
(*if t* = *a then* (*if ka* = *t then 1*::′*a else* (*0*::′*a*)) ∗ *q else if ka* = *t then 1*::′*a else* (*0*::′*a*))) = (*1*::′*a*)
proof −
let *?f*= λ*ka*. (*if ka* = *a then* (*if t* = *ka then 1*::′*a else* (*0*::′*a*)) ∗ *k else if t* = *ka then 1*::′*a else* (*0*::′*a*)) ∗
(*if t* = *a then* (*if ka* = *t then 1*::′*a else* (*0*::′*a*)) ∗ *q else if ka* = *t then 1*::′*a else* (*0*::′*a*))
have *univ-rw*:*UNIV* = (*UNIV*−{*t*}) ∪ {*t*} **by** *auto*
have *setsum ?f UNIV* = *setsum ?f* ((*UNIV*−{*t*}) ∪ {*t*}) **using** *univ-rw* **by** *auto*

also have ... = *setsum ?f* (*UNIV*−{*t*}) + *setsum ?f* {*t*} **by** (*rule setsum-Un-disjoint*, *auto*)
also have ... = *setsum ?f* {*t*} **by** *auto*
also have ... = *1* **using** *kq* **by** *auto*
finally show *?thesis* **.**
qed
fix *s* **assume** *s-not-t*: $s \neq t$
show ($\sum$ *ka*∈*UNIV*. (*if ka* = *a then* (*if s* = *ka then 1*::′*a else* (*0*::′*a*)) ∗ *k else if s* = *ka then 1*::′*a else* (*0*::′*a*)) ∗
(*if t* = *a then* (*if ka* = *t then 1*::′*a else* (*0*::′*a*)) ∗ *q else if ka* = *t then 1*::′*a else* (*0*::′*a*))) = *0*
apply (*rule setsum-0*′) **using** *s-not-t* **by** *auto*
qed
qed

corollary *invertible-mult-column*′:
assumes *q-not-zero*: $q \neq 0$
shows *invertible* (*mult-column* (*mat* (*1*::′*a*::{*field*})) *a q*)
by (*simp add*: *invertible-mult-column*[*of q inverse q*] *q-not-zero*)

Properties about *column-add*

lemma *column-add-mat-1*: *A* ∗∗ *column-add* (*mat 1*) *a b q* = *column-add A a b q*
proof (*unfold matrix-matrix-mult-def*,
unfold column-add-def, *vector*, *auto*)
fix *i*
let *?f*=λ*k*. *A* $ *i* $ *k* ∗ (*mat* (*1*::′*a*) $ *k* $ *a* + *mat* (*1*::′*a*) $ *k* $ *b* ∗ *q*)
show *setsum ?f UNIV* = *A* $ *i* $ *a* + *A* $ *i* $ *b* ∗ *q*
proof (*cases a*=*b*)
case *True*
have *univ-rw*:*UNIV* = (*UNIV*−{*a*}) ∪ {*a*} **by** *auto*
have *setsum ?f UNIV* = *setsum ?f* ((*UNIV*−{*a*}) ∪ {*a*}) **using** *univ-rw* **by** *auto*
also have ... = *setsum ?f* (*UNIV*−{*a*}) + *setsum ?f* {*a*} **by** (*rule setsum-Un-disjoint*, *auto*)
also have ... = *setsum ?f* {*a*} **unfolding** *mat-def True* **by** *auto*
also have ... = *?f a* **by** *auto*
also have ... = *A* $ *i* $ *a* + *A* $ *i* $ *b* ∗ *q* **using** *True* **unfolding** *mat-1-fun* **using** *distrib-left*[*of A* $ *i* $ *b 1 q*] **by** *auto*
finally show *?thesis* **.**
next
case *False*
have *univ-rw*: *UNIV* = {*a*} ∪ ({*b*} ∪ (*UNIV* − {*a*} − {*b*})) **by** *auto*
have *setsum-rw*: *setsum ?f* ({*b*} ∪ (*UNIV* − {*a*} − {*b*})) = *setsum ?f* {*b*} + *setsum ?f* (*UNIV* − {*a*} − {*b*}) **by** (*rule setsum-Un-disjoint*, *auto simp add*: *False*)
have *setsum ?f UNIV* = *setsum ?f* ({*a*} ∪ ({*b*} ∪ (*UNIV* − {*a*} − {*b*}))) **using** *univ-rw* **by** *simp*
also have ... = *setsum ?f* {*a*} + *setsum ?f* ({*b*} ∪ (*UNIV* − {*a*} − {*b*})) **by** (*rule setsum-Un-disjoint*, *auto simp add*: *False*)

also have ... = *setsum ?f* $\{a\}$ + *setsum ?f* $\{b\}$ + *setsum ?f* (*UNIV* − $\{a\}$ − $\{b\}$) **unfolding** *setsum-rw ab-semigroup-add-class.add-ac*(*1*)[*symmetric*] **..**
also have ... = *setsum ?f* $\{a\}$ + *setsum ?f* $\{b\}$ **unfolding** *mat-def* **by** *auto*

also have ... = *A* $ *i* $ *a* + *A* $ *i* $ *b* ∗ *q* **using** *False* **unfolding** *mat-def* **by** *simp*
finally show *?thesis* **.**
qed
fix *j*
assume *j-noteq-a*: *j*≠*a*
show (∑*k*∈*UNIV*. *A* $ *i* $ *k* ∗ *mat* (*1*::′*a*) $ *k* $ *j*) = *A* $ *i* $ *j*
proof −
let *?f*=λ*k*. *A* $ *i* $ *k* ∗ *mat* (*1*::′*a*) $ *k* $ *j*
have *univ-rw*:*UNIV* = (*UNIV*−$\{j\}$) ∪ $\{j\}$ **by** *auto*
have *setsum ?f UNIV* = *setsum ?f* ((*UNIV*−$\{j\}$) ∪ $\{j\}$) **using** *univ-rw* **by** *auto*
also have ... = *setsum ?f* (*UNIV*−$\{j\}$) + *setsum ?f* $\{j\}$ **by** (*rule setsum-Un-disjoint*, *auto*)
also have ... = *setsum ?f* $\{j\}$ **unfolding** *mat-def* **by** *auto*
also have ... = *A* $ *i* $ *j* **unfolding** *mat-def* **by** *simp*
finally show *?thesis* **.**
qed
qed

lemma *invertible-column-add*:
assumes *a-noteq-b*: *a*≠*b*
shows *invertible* (*column-add* (*mat* (*1*::′*a*::$\{$*ring-1*$\}$)) *a b q*)
proof (*unfold invertible-def*, *rule exI*[*of* - (*column-add* (*mat 1*) *a b* (−*q*))], *rule conjI*)
show *column-add* (*mat* (*1*::′*a*)) *a b q* ∗∗ *column-add* (*mat* (*1*::′*a*)) *a b* (− *q*) = *mat* (*1*::′*a*) **using** *a-noteq-b*
proof (*unfold matrix-matrix-mult-def*, *vector*, *clarify*, *unfold column-add-def*, *vector*, *unfold mat-1-fun*, *auto*)
show (∑*k*∈*UNIV*. (*if k* = *a then* (*0*::′*a*) + (*1*::′*a*) ∗ *q else if b* = *k then 1*::′*a else* (*0*::′*a*)) ∗ (*if k* = *b then 1*::′*a else* (*0*::′*a*))) = (*1*::′*a*)
proof −
let *?f*=λ*k*. (*if k* = *a then* (*0*::′*a*) + (*1*::′*a*) ∗ *q else if b* = *k then 1*::′*a else* (*0*::′*a*)) ∗ (*if k* = *b then 1*::′*a else* (*0*::′*a*))
have *univ-rw*:*UNIV* = (*UNIV*−$\{b\}$) ∪ $\{b\}$ **by** *auto*
have *setsum ?f UNIV* = *setsum ?f* ((*UNIV*−$\{b\}$) ∪ $\{b\}$) **using** *univ-rw* **by** *auto*
also have ... = *setsum ?f* (*UNIV*−$\{b\}$) + *setsum ?f* $\{b\}$ **by** (*rule setsum-Un-disjoint*, *auto*)
also have ... = *setsum ?f* $\{b\}$ **by** *auto*
also have ... = *1* **using** *a-noteq-b* **by** *simp*
finally show *?thesis* **.**
qed
show (∑*k*∈*UNIV*. (*if k* = *a then* (*1*::′*a*) + (*0*::′*a*) ∗ *q else if a* = *k then 1*::′*a*

else $(0::'a)) * ((if\ k = a\ then\ 1::'a\ else\ (0::'a)) + - ((if\ k = b\ then\ 1::'a\ else\ (0::'a)) * q))) =$
$(1::'a)$
proof −
let $?f=\lambda k.\ (if\ k = a\ then\ (1::'a) + (0::'a) * q\ else\ if\ a = k\ then\ 1::'a\ else\ (0::'a)) * ((if\ k = a\ then\ 1::'a\ else\ (0::'a)) + - ((if\ k = b\ then\ 1::'a\ else\ (0::'a)) * q))$
have *univ-rw*: $UNIV = (UNIV - \{a\}) \cup \{a\}$ **by** *auto*
have *setsum ?f UNIV = setsum ?f* $((UNIV - \{a\}) \cup \{a\})$ **using** *univ-rw* **by** *auto*
also have ... = *setsum ?f* $(UNIV - \{a\})$ + *setsum ?f* $\{a\}$ **by** (*rule setsum-Un-disjoint*, *auto*)
also have ... = *setsum ?f* $\{a\}$ **by** *auto*
also have ... = *1* **using** *a-noteq-b* **by** *simp*
finally show *?thesis* **.**
qed
fix *i j*
assume *i-not-b*: $i \neq b$ **and** *i-not-a*: $i \neq a$ **and** *i-not-j*: $i \neq j$
show $(\sum k \in UNIV.\ (if\ k = a\ then\ (0::'a) + (0::'a) * q\ else\ if\ i = k\ then\ 1::'a\ else\ (0::'a)) *$
$(if\ j = a\ then\ (if\ k = a\ then\ 1::'a\ else\ (0::'a)) + (if\ k = b\ then\ 1::'a\ else\ (0::'a)) * - q\ else\ if\ k = j\ then\ 1::'a\ else\ (0::'a))) = (0::'a)$
by (*rule setsum-0'*, *auto simp add*: *i-not-b i-not-a i-not-j*)
next
fix *j*
assume *a-not-j*: $a \neq j$
show $(\sum k \in UNIV.\ (if\ k = a\ then\ (1::'a) + (0::'a) * q\ else\ if\ a = k\ then\ 1::'a\ else\ (0::'a)) * (if\ k = j\ then\ 1::'a\ else\ (0::'a))) = (0::'a)$
apply (*rule setsum-0'*) **using** *a-not-j a-noteq-b* **by** *auto*
next
fix *j*
assume *j-not-b*: $j \neq b$ **and** *j-not-a*: $j \neq a$
show $(\sum k \in UNIV.\ (if\ k = a\ then\ (0::'a) + (0::'a) * q\ else\ if\ j = k\ then\ 1::'a\ else\ (0::'a)) * (if\ k = j\ then\ 1::'a\ else\ (0::'a))) = (1::'a)$
proof −
let $?f=\lambda k.\ (if\ k = a\ then\ (0::'a) + (0::'a) * q\ else\ if\ j = k\ then\ 1::'a\ else\ (0::'a)) * (if\ k = j\ then\ 1::'a\ else\ (0::'a))$
have *univ-rw*: $UNIV = (UNIV - \{j\}) \cup \{j\}$ **by** *auto*
have *setsum ?f UNIV = setsum ?f* $((UNIV - \{j\}) \cup \{j\})$ **using** *univ-rw* **by** *auto*
also have ... = *setsum ?f* $(UNIV - \{j\})$ + *setsum ?f* $\{j\}$ **by** (*rule setsum-Un-disjoint*, *auto*)
also have ... = *setsum ?f* $\{j\}$ **using** *j-not-b j-not-a* **by** *auto*
also have ... = *1* **using** *j-not-b j-not-a* **by** *auto*
finally show *?thesis* **.**
qed
next
fix *j*
assume *b-not-j*: $b \neq j$

show ($\sum k \in UNIV$. (*if* $k = a$ *then* $0 + 1 * q$ *else if* $b = k$ *then* 1 *else* 0) $*$
(*if* $j = a$ *then* (*if* $k = a$ *then* 1 *else* 0) $+$ (*if* $k = b$ *then* 1 *else* 0) $* - q$ *else if* $k = j$ *then* 1 *else* 0)) $= 0$
proof (*cases* $j=a$)
case *False*
show *?thesis* **by** (*rule setsum-0'*, *auto simp add*: *False b-not-j*)
next
case *True* — This case is different from the other cases
let $?f=\lambda k$. (*if* $k = a$ *then* $0 + 1 * q$ *else if* $b = k$ *then* 1 *else* 0) $*$
(*if* $j = a$ *then* (*if* $k = a$ *then* 1 *else* 0) $+$ (*if* $k = b$ *then* 1 *else* 0) $* - q$ *else if* $k = j$ *then* 1 *else* 0)
have *univ-eq*: $UNIV = ((UNIV - \{a\} - \{b\}) \cup (\{b\} \cup \{a\}))$ **by** *auto*
have *setsum-a*: *setsum* $?f$ $\{a\} = q$ **unfolding** *True* **using** *b-not-j* **using** *a-noteq-b* **by** *auto*
have *setsum-b*: *setsum* $?f$ $\{b\} = -q$ **unfolding** *True* **using** *b-not-j* **using** *a-noteq-b* **by** *auto*
have *setsum-rest*: *setsum* $?f$ $(UNIV - \{a\} - \{b\}) = 0$ **by** (*rule setsum-0'*, *auto simp add*: *True b-not-j a-noteq-b*)
have *setsum* $?f$ $UNIV$ $=$ *setsum* $?f$ $((UNIV - \{a\} - \{b\}) \cup (\{b\} \cup \{a\}))$ **using** *univ-eq* **by** *simp*
also have ... $=$ *setsum* $?f$ $(UNIV - \{a\} - \{b\})$ $+$ *setsum* $?f$ $(\{b\} \cup \{a\})$ **by** (*rule setsum-Un-disjoint*, *auto*)
also have ... $=$ *setsum* $?f$ $(UNIV - \{a\} - \{b\})$ $+$ *setsum* $?f$ $\{b\}$ $+$ *setsum* $?f$ $\{a\}$ **by** (*auto simp add*: *setsum-Un-disjoint a-noteq-b*)
also have ... $= 0$ **unfolding** *setsum-a setsum-b setsum-rest* **by** *simp*
finally show *?thesis* **.**
qed
qed
next
show *column-add* (*mat* $(1::'a)$) a b $(-q)$ $**$ *column-add* (*mat* $(1::'a)$) a b q $=$ *mat* $(1::'a)$ **using** *a-noteq-b*
proof (*unfold matrix-matrix-mult-def*, *vector*, *clarify*, *unfold column-add-def*, *vector*, *unfold mat-1-fun*, *auto*)
show ($\sum k \in UNIV$. (*if* $k = a$ *then* $(0::'a) + (1::'a) * - q$ *else if* $b = k$ *then* $1::'a$ *else* $(0::'a)$) $*$ (*if* $k = b$ *then* $1::'a$ *else* $(0::'a)$)) $= (1::'a)$
proof $-$
let $?f=\lambda k$. (*if* $k = a$ *then* $(0::'a) + (1::'a) * - q$ *else if* $b = k$ *then* $1::'a$ *else* $(0::'a)$) $*$ (*if* $k = b$ *then* $1::'a$ *else* $(0::'a)$)
have *univ-rw*:$UNIV = (UNIV - \{b\}) \cup \{b\}$ **by** *auto*
have *setsum* $?f$ $UNIV$ $=$ *setsum* $?f$ $((UNIV - \{b\}) \cup \{b\})$ **using** *univ-rw* **by** *auto*
also have ... $=$ *setsum* $?f$ $(UNIV - \{b\})$ $+$ *setsum* $?f$ $\{b\}$ **by** (*rule setsum-Un-disjoint*, *auto*)
also have ... $=$ *setsum* $?f$ $\{b\}$ **by** *auto*
also have ... $=$ 1 **using** *a-noteq-b* **by** *auto*
finally show *?thesis* **.**
qed
next
show ($\sum k \in UNIV$. (*if* $k = a$ *then* $(1::'a) + (0::'a) * - q$ *else if* $a = k$ *then*

1::'a else (0::'a)) ∗ ((if k = a then 1::'a else (0::'a)) + (if k = b then 1::'a else (0::'a)) ∗ q)) =
(1::'a)
proof −
let *?f=λk. (if k = a then (1::'a) + (0::'a) ∗ − q else if a = k then 1::'a else (0::'a)) ∗ ((if k = a then 1::'a else (0::'a)) + (if k = b then 1::'a else (0::'a)) ∗ q)*
have *univ-rw:UNIV = (UNIV−{a}) ∪ {a}* **by** *auto*
have *setsum ?f UNIV = setsum ?f ((UNIV−{a}) ∪ {a})* **using** *univ-rw* **by** *auto*
also have ... = *setsum ?f (UNIV−{a}) + setsum ?f {a}* **by** (*rule setsum-Un-disjoint*, *auto*)
also have ... = *setsum ?f {a}* **by** *auto*
also have ... = *1* **using** *a-noteq-b* **by** *auto*
finally show *?thesis* **.**
qed
next
fix *j*
assume *a-not-j*: *a ≠ j* **show** (∑*k∈UNIV. (if k = a then (1::'a) + (0::'a) ∗ − q else if a = k then 1::'a else (0::'a)) ∗ (if k = j then 1::'a else (0::'a))) = (0::'a)*
apply (*rule setsum-0'*) **using** *a-not-j* **by** *auto*
next
fix *j*
assume *j-not-b*: *j ≠ b* **and** *j-not-a*: *j ≠ a*
show (∑*k∈UNIV. (if k = a then (0::'a) + (0::'a) ∗ − q else if j = k then 1::'a else (0::'a)) ∗ (if k = j then 1::'a else (0::'a))) = (1::'a)*
proof −
let *?f=λk.(if k = a then (0::'a) + (0::'a) ∗ − q else if j = k then 1::'a else (0::'a)) ∗ (if k = j then 1::'a else (0::'a))*
have *univ-rw:UNIV = (UNIV−{j}) ∪ {j}* **by** *auto*
have *setsum ?f UNIV = setsum ?f ((UNIV−{j}) ∪ {j})* **using** *univ-rw* **by** *auto*
also have ... = *setsum ?f (UNIV−{j}) + setsum ?f {j}* **by** (*rule setsum-Un-disjoint*, *auto*)
also have ... = *setsum ?f {j}* **by** *auto*
also have ... = *1* **using** *a-noteq-b j-not-b j-not-a* **by** *auto*
finally show *?thesis* **.**
qed
next
fix *i j*
assume *i-not-b*: *i ≠ b* **and** *i-not-a*: *i ≠ a* **and** *i-not-j*: *i ≠ j*
show (∑*k∈UNIV. (if k = a then (0::'a) + (0::'a) ∗ − q else if i = k then 1::'a else (0::'a)) ∗*
(if j = a then (if k = a then 1::'a else (0::'a)) + (if k = b then 1::'a else (0::'a)) ∗ q else if k = j then 1::'a else (0::'a))) = (0::'a)
by (*rule setsum-0'*, *auto simp add*: *i-not-b i-not-a i-not-j*)
next
fix *j*
assume *b-not-j*: *b ≠ j*

show ($\sum k \in UNIV$. (*if* $k = a$ *then* $(0::'a) + (1::'a) * - q$ *else if* $b = k$ *then* $1::'a$ *else* $(0::'a)) *$
(*if* $j = a$ *then* (*if* $k = a$ *then* $1::'a$ *else* $(0::'a)$) + (*if* $k = b$ *then* $1::'a$ *else* $(0::'a)) * q$ *else if* $k = j$ *then* $1::'a$ *else* $(0::'a)$)) $= 0$
proof (*cases* $j=a$)
case *False*
show *?thesis* **by** (*rule setsum-0'*, *auto simp add*: *False b-not-j*)
next
case *True* — This case is different from the other cases
let $?f=\lambda k$. (*if* $k = a$ *then* $(0::'a) + (1::'a) * - q$ *else if* $b = k$ *then* $1::'a$ *else* $(0::'a)) *$
(*if* $j = a$ *then* (*if* $k = a$ *then* $1::'a$ *else* $(0::'a)$) + (*if* $k = b$ *then* $1::'a$ *else* $(0::'a)) * q$ *else if* $k = j$ *then* $1::'a$ *else* $(0::'a)$)
have *univ-eq*: $UNIV = ((UNIV - \{a\} - \{b\}) \cup (\{b\} \cup \{a\}))$ **by** *auto*
have *setsum-a*: *setsum* $?f\ \{a\} = -q$ **unfolding** *True* **using** *b-not-j* **using** *a-noteq-b* **by** *auto*
have *setsum-b*: *setsum* $?f\ \{b\} = q$ **unfolding** *True* **using** *b-not-j* **using** *a-noteq-b* **by** *auto*
have *setsum-rest*: *setsum* $?f\ (UNIV - \{a\} - \{b\}) = 0$ **by** (*rule setsum-0'*, *auto simp add*: *True b-not-j a-noteq-b*)
have *setsum* $?f\ UNIV =$ *setsum* $?f\ ((UNIV - \{a\} - \{b\}) \cup (\{b\} \cup \{a\}))$ **using** *univ-eq* **by** *simp*
also have ... = *setsum* $?f\ (UNIV - \{a\} - \{b\})$ + *setsum* $?f\ (\{b\} \cup \{a\})$ **by** (*rule setsum-Un-disjoint*, *auto*)
also have ... = *setsum* $?f\ (UNIV - \{a\} - \{b\})$ + *setsum* $?f\ \{b\}$ + *setsum* $?f\ \{a\}$ **by** (*auto simp add*: *setsum-Un-disjoint a-noteq-b*)
also have ... $= 0$ **unfolding** *setsum-a setsum-b setsum-rest* **by** *simp*
finally show *?thesis* **.**
qed
qed
qed

Relationships between *interchange-rows* and *interchange-columns*

lemma *interchange-rows-transpose*:
shows *interchange-rows* (*transpose A*) *a b* = *transpose* (*interchange-columns A a b*)
unfolding *interchange-rows-def interchange-columns-def transpose-def* **by** *vector*

lemma *interchange-rows-transpose'*:
shows *interchange-rows A a b* = *transpose* (*interchange-columns* (*transpose A*) *a b*)
unfolding *interchange-rows-def interchange-columns-def transpose-def* **by** *vector*

lemma *interchange-columns-transpose*:
shows *interchange-columns* (*transpose A*) *a b* = *transpose* (*interchange-rows A a b*)
unfolding *interchange-rows-def interchange-columns-def transpose-def* **by** *vector*

lemma *interchange-columns-transpose'*:

shows *interchange-columns A a b* = *transpose* (*interchange-rows* (*transpose A*) *a b*)
unfolding *interchange-rows-def interchange-columns-def transpose-def* **by** *vector*

Code equations for *interchange-rows ?A ?a ?b* = (χ*i j. if i = ?a then ?A* \$ *?b* \$ *j else if i = ?b then ?A* \$ *?a* \$ *j else ?A* \$ *i* \$ *j*), *interchange-columns ?A ?n ?m* = (χ*i j. if j = ?n then ?A* \$ *i* \$ *?m else if j = ?m then ?A* \$ *i* \$ *?n else ?A* \$ *i* \$ *j*), *row-add ?A ?a ?b ?q* = (χ*i j. if i = ?a then ?A* \$ *?a* \$ *j + ?q * ?A* \$ *?b* \$ *j else ?A* \$ *i* \$ *j*), *column-add ?A ?n ?m ?q* = (χ*i j. if j = ?n then ?A* \$ *i* \$ *?n + ?A* \$ *i* \$ *?m * ?q else ?A* \$ *i* \$ *j*), *mult-row ?A ?a ?q* = (χ*i j. if i = ?a then ?q * ?A* \$ *?a* \$ *j else ?A* \$ *i* \$ *j*) and *mult-column ?A ?n ?q* = (χ*i j. if j = ?n then ?A* \$ *i* \$ *j * ?q else ?A* \$ *i* \$ *j*):

definition *interchange-rows-row*
where *interchange-rows-row A a b i* = *vec-lambda* (%*j. if i = a then A* \$ *b* \$ *j else if i = b then A* \$ *a* \$ *j else A* \$ *i* \$ *j*)

lemma *interchange-rows-code* [*code abstract*]:
vec-nth (*interchange-rows-row A a b i*) = (%*j. if i = a then A* \$ *b* \$ *j else if i = b then A* \$ *a* \$ *j else A* \$ *i* \$ *j*)
unfolding *interchange-rows-row-def* **by** *auto*

lemma *interchange-rows-code-nth* [*code abstract*]: *vec-nth* (*interchange-rows A a b*) = *interchange-rows-row A a b*
unfolding *interchange-rows-def* **unfolding** *interchange-rows-row-def*[*abs-def*]
by *auto*

definition *interchange-columns-row*
where *interchange-columns-row A n m i* = *vec-lambda* (%*j. if j = n then A* \$ *i* \$ *m else if j = m then A* \$ *i* \$ *n else A* \$ *i* \$ *j*)

lemma *interchange-columns-code* [*code abstract*]:
vec-nth (*interchange-columns-row A n m i*) = (%*j. if j = n then A* \$ *i* \$ *m else if j = m then A* \$ *i* \$ *n else A* \$ *i* \$ *j*)
unfolding *interchange-columns-row-def* **by** *auto*

lemma *interchange-columns-code-nth* [*code abstract*]: *vec-nth* (*interchange-columns A a b*) = *interchange-columns-row A a b*
unfolding *interchange-columns-def* **unfolding** *interchange-columns-row-def*[*abs-def*]
by *auto*

definition *row-add-row*
where *row-add-row A a b q i* = *vec-lambda* (%*j. if i = a then A* \$ *a* \$ *j + q * A* \$ *b* \$ *j else A* \$ *i* \$ *j*)

lemma *row-add-code* [*code abstract*]:
vec-nth (*row-add-row A a b q i*) = (%*j. if i = a then A* \$ *a* \$ *j + q * A* \$ *b* \$ *j else A* \$ *i* \$ *j*)

unfolding *row-add-row-def* **by** *auto*

lemma *row-add-code-nth* [*code abstract*]: *vec-nth* (*row-add A a b q*) = *row-add-row A a b q*
unfolding *row-add-def* **unfolding** *row-add-row-def*[*abs-def*]
by *auto*

definition *column-add-row*
where *column-add-row A n m q i* = *vec-lambda* (%*j*. *if j* = *n then A* $ *i* $ *n* + *A* $ *i* $ *m* * *q else A* $ *i* $ *j*)

lemma *column-add-code* [*code abstract*]:
vec-nth (*column-add-row A n m q i*) = (%*j*. *if j* = *n then A* $ *i* $ *n* + *A* $ *i* $ *m* * *q else A* $ *i* $ *j*)
unfolding *column-add-row-def* **by** *auto*

lemma *column-add-code-nth* [*code abstract*]: *vec-nth* (*column-add A a b q*) = *column-add-row A a b q*
unfolding *column-add-def* **unfolding** *column-add-row-def*[*abs-def*]
by *auto*

definition *mult-row-row*
where *mult-row-row A a q i* = *vec-lambda* (%*j*. *if i* = *a then q* * *A* $ *a* $ *j else A* $ *i* $ *j*)

lemma *mult-row-code* [*code abstract*]:
vec-nth (*mult-row-row A a q i*) = (%*j*. *if i* = *a then q* * *A* $ *a* $ *j else A* $ *i* $ *j*)
unfolding *mult-row-row-def* **by** *auto*

lemma *mult-row-code-nth* [*code abstract*]: *vec-nth* (*mult-row A a q*) = *mult-row-row A a q*
unfolding *mult-row-def* **unfolding** *mult-row-row-def*[*abs-def*]
by *auto*

definition *mult-column-row*
where *mult-column-row A n q i* = *vec-lambda* (%*j*. *if j* = *n then A* $ *i* $ *j* * *q else A* $ *i* $ *j*)

lemma *mult-column-code* [*code abstract*]:
vec-nth (*mult-column-row A n q i*) = (%*j*. *if j* = *n then A* $ *i* $ *j* * *q else A* $ *i* $ *j*)
unfolding *mult-column-row-def* **by** *auto*

lemma *mult-column-code-nth* [*code abstract*]: *vec-nth* (*mult-column A a q*) = *mult-column-row A a q*
unfolding *mult-column-def* **unfolding** *mult-column-row-def*[*abs-def*]
by *auto*

Definitions of row space, column space and rank

definition *col-space* :: $'a$::{*real-vector*}^$'n$^$'m$=>($'a$^$'m$) *set*
  where *col-space A* = *span* (*columns A*)

definition *row-space* :: $'a$::{*real-vector*}^$'n$^$'m$=>($'a$^$'n$) *set*
  where *row-space A* = *span* (*rows A*)

definition *row-rank* :: $'a$::{*real-vector*}^$'n$^$'m$=>*nat*
  where *row-rank A* = *dim* (*row-space A*)

definition *col-rank* :: $'a$::{*real-vector*}^$'n$^$'m$=>*nat*
  where *col-rank A* = *dim* (*col-space A*)

definition *rank* :: *real*^$'n$^$'m$=>*nat*
  where *rank A* = *row-rank A*

lemma *matrix-vmult-column-sum*:
  fixes *A*::*real*^$'n$^$'m$
  shows ∃*f*. *A* ∗*v* *x* = *setsum* (λ*y*. *f y* $*_R$ *y*) (*columns A*)
proof (*rule exI*[*of* - λ*y*. *setsum* (λ*i*. *x* \$ *i*) {*i*. *y* = *column i A*}])
  let *?f*=λ*y*. *setsum* (λ*i*. *x* \$ *i*) {*i*. *y* = *column i A*}
  let *?g*=(λ*y*. {*i*. *y*=*column i* (*A*)})
  have *inj*: *inj-on ?g* (*columns* (*A*)) **unfolding** *inj-on-def* **unfolding** *columns-def* **by** *auto*
  have *union-univ*: ⋃ (*?g*‘(*columns* (*A*))) = *UNIV* **unfolding** *columns-def* **by** *auto*
  have *A* ∗*v* *x* = (∑*i*∈*UNIV*. *x* \$ *i* ∗*s* *column i A*) **unfolding** *matrix-mult-vsum* **..**
  also have ... = *setsum* (λ*i*. *x* \$ *i* ∗*s* *column i A*) (⋃(*?g*‘(*columns A*))) **unfolding** *union-univ* **..**
  also have ... = *setsum* (*setsum* ((λ*i*. *x* \$ *i* ∗*s* *column i A*))) (*?g*‘(*columns A*)) **by** (*rule setsum-Union-disjoint*, *auto*)
  also have ... = *setsum* ((*setsum* ((λ*i*. *x* \$ *i* ∗*s* *column i A*))) ∘ *?g*) (*columns A*) **by** (*rule setsum-reindex*, *simp add*: *inj*)
  also have ... = *setsum* (λ*y*. *?f y* $*_R$ *y*) (*columns A*)
  proof (*rule setsum-cong2*, *unfold o-def*)
    fix *xa*
    have *setsum* (λ*i*. *x* \$ *i* ∗*s* *column i A*) {*i*. *xa* = *column i A*} = *setsum* (λ*i*. *x* \$ *i* ∗*s* *xa*) {*i*. *xa* = *column i A*} **by** *simp*
    also have ... = *setsum* (λ*i*. *x* \$ *i* $*_R$ *xa*) {*i*. *xa* = *column i A*} **unfolding** *scalar-mult-eq-scaleR* **..**
    also have ... = *setsum* (λ*i*. *x* \$ *i*) {*i*. *xa* = *column i A*} $*_R$ *xa* **using** *scaleR-setsum-left*[*of* (λ*i*. *x* \$ *i*) {*i*. *xa* = *column i A*} *xa*] **..**
    finally show (∑*i* | *xa* = *column i A*. *x* \$ *i* ∗*s* *column i A*) = (∑*i* | *xa* = *column i A*. *x* \$ *i*) $*_R$ *xa* **.**
  qed
  finally show *A* ∗*v* *x* = (∑*y*∈*columns A*. (∑*i* | *y* = *column i A*. *x* \$ *i*) $*_R$ *y*) **.**
qed

lemma *row-space-eq-col-space-transpose*:

fixes $A::'a::\{real\text{-}vector,\ semiring\text{-}1\}$^$'columns$^$'rows$
shows *row-space* A = *col-space* (*transpose* A)
unfolding *col-space-def row-space-def columns-transpose*[*of* A] **..**

lemma *col-space-eq*:
fixes A::*real*^$'m$^$'n$
shows *col-space* A = *Dim-Formula.col-space* A
proof (*unfold col-space-def col-space-eq span-explicit*, *rule*)
show $\{y.\ \exists x.\ A *v\ x = y\} \subseteq \{y.\ \exists S\ u.\ finite\ S \wedge S \subseteq columns\ A \wedge (\sum v{\in}S.\ u\ v *_R v) = y\}$
proof (*rule*)
fix y **assume** y: $y \in \{y.\ \exists x.\ A *v\ x = y\}$
obtain x **where** $x{:}A *v\ x = y$ **using** y **by** *blast*
obtain f **where** *setsum*: $A *v\ x$ = *setsum* $(\lambda y.\ f\ y *_R y)$ (*columns* A) **using** *matrix-vmult-column-sum* **by** *auto*
have *finite-cols*: *finite* (*columns* A)
proof −
def $f{=}{=}\lambda i.$ *column* i A
show *?thesis* **unfolding** *columns-def* **using** *finite-Atleast-Atmost-nat*[*of* f] **unfolding** *f-def* **by** *simp*
qed
show $y \in \{y.\ \exists S\ u.\ finite\ S \wedge S \subseteq columns\ A \wedge (\sum v{\in}S.\ u\ v *_R v) = y\}$
by (*rule*, *rule exI*[*of* - *columns* A], *auto*, *rule finite-cols*, *rule exI*[*of* - f], *metis* x *setsum*)
qed
next
show $\{y.\ \exists S\ u.\ finite\ S \wedge S \subseteq columns\ A \wedge (\sum v{\in}S.\ u\ v *_R v) = y\} \subseteq \{y.\ \exists x.\ A *v\ x = y\}$
proof (*rule*, *auto*)
fix S **and** u::(*real*, $'n$) *vec* $\Rightarrow$ *real*
assume *finite-S*: *finite* S **and** *S-in-cols*: $S \subseteq$ *columns* A
let $?f{=}\lambda x.$ *if* $x \in S$ *then* $u\ x$ *else* 0
let $?x = (\chi\ i.$ *if column* i $A \in S$ *then* (*inverse* (*real* (*card* $\{a.$ *column* i $A{=}$*column* a $A\}$))) $*$ u (*column* i A) *else* 0)
show $\exists x.\ A *v\ x = (\sum v{\in}S.\ u\ v *_R v)$
proof (*unfold matrix-mult-vsum*, *rule exI*[*of* - *?x*], *simp*)
let $?g{=}\lambda y.\ \{i.\ y{=}column\ i\ A\}$
have *inj*: *inj-on ?g* (*columns* A) **unfolding** *inj-on-def* **unfolding** *columns-def* **by** *auto*
have *union-univ*: $\bigcup(?g`(columns\ A)) = UNIV$ **unfolding** *columns-def* **by** *auto*
have *setsum* ($\lambda i.$(*if column* i $A \in S$ *then inverse* (*real* (*card* $\{a.$ *column* i A = *column* a $A\}$)) $*$ u (*column* i A) *else* 0) $*s$ *column* i A) *UNIV*
= *setsum* ($\lambda i.$(*if column* i $A \in S$ *then inverse* (*real* (*card* $\{a.$ *column* i A = *column* a $A\}$)) $*$ u (*column* i A) *else* 0) $*s$ *column* i A) $(\bigcup(?g`(columns\ A)))$
unfolding *union-univ* **..**
also have ... = *setsum* (*setsum* ($\lambda i.$(*if column* i $A \in S$ *then inverse* (*real* (*card* $\{a.$ *column* i A = *column* a $A\}$)) $*$ u (*column* i A) *else* 0) $*s$ *column* i A)) $(?g`(columns\ A))$

by (*rule setsum-Union-disjoint*, *auto*)
also have ... = *setsum* ((*setsum* (λ*i*.(*if column i A* $\in$ *S then inverse* (*real* (*card* {*a*. *column i A* = *column a A*})) $*$ *u* (*column i A*) *else 0*) $*s$ *column i A*)) $\circ$ *?g*)
(*columns A*) **by** (*rule setsum-reindex*, *simp add*: *inj*)
also have ... = *setsum* (λ*y*. *?f y* $*_R$ *y*) (*columns A*)
proof (*rule setsum-cong2*, *auto*)
fix *x*
assume *x-in-cols*: *x* $\in$ *columns A* **and** *x-notin-S*: *x* $\notin$ *S*
show ($\sum$ *i* | *x* = *column i A*. (*if column i A* $\in$ *S then* (*inverse* (*real* (*card* {*a*. *column i A* = *column a A*}))) $*$ *u* (*column i A*) *else 0*) $*s$ *column i A*) = *0*
apply (*rule setsum-0'*) **using** *x-notin-S* **by** *auto*
next
fix *x*
assume *x-in-cols*: *x* $\in$ *columns A* **and** *x-in-S*: *x* $\in$ *S*
have *setsum* (λ*i*. (*if column i A* $\in$ *S then* (*inverse* (*real* (*card* {*a*. *column i A* = *column a A*}))) $*$ *u* (*column i A*) *else 0*) $*s$ *column i A*) {*i*. *x* = *column i A*} =
setsum (λ*i*. (*inverse* (*real* (*card* {*a*. *column i A* = *column a A*}))) $*$ *u* (*column i A*) $*s$ *column i A*) {*i*. *x* = *column i A*} **apply** (*rule setsum-cong2*) **using** *x-in-S* **by** *simp*
also have ... = *setsum* (λ*i*. (*inverse* (*real* (*card* {*a*. *x* = *column a A*}))) $*$ *u x* $*s$ *x*) {*i*. *x* = *column i A*} **by** *auto*
also have ... = *of-nat* (*card* {*i*. *x* = *column i A*}) $*$ (*inverse* (*real* (*card* {*a*. *x* = *column a A*})) $*$ *u x* $*s$ *x*) **unfolding** *setsum-constant* **..**
also have ... = *of-nat* (*card* {*i*. *x* = *column i A*}) $*$ (*inverse* (*real* (*card* {*a*. *x* = *column a A*})) $*s$ (*u x* $*s$ *x*))
unfolding *vector-smult-assoc* [*of inverse* (*real* (*card* {*a*. *x* = *column a A*})) *u x x* , *symmetric*] **..**
also have ... = *real* (*card* {*i*. *x* = *column i A*}) $*_R$ (*inverse* (*real* (*card* {*a*. *x* = *column a A*})) $*s$ (*u x* $*s$ *x*))
by (*auto*, *metis real-of-nat-def setsum-constant setsum-constant-scaleR*)
also have ... = *real* (*card* {*i*. *x* = *column i A*}) $*s$ (*inverse* (*real* (*card* {*a*. *x* = *column a A*})) $*s$ (*u x* $*s$ *x*)) **unfolding** *scalar-mult-eq-scaleR* **..**
also have ... = (*real* (*card* {*i*. *x* = *column i A*}) $*$ *inverse* (*real* (*card* {*a*. *x* = *column a A*}))) $*s$ (*u x* $*s$ *x*) **unfolding** *vector-smult-assoc* **by** *auto*
also have ... = *1* $*s$ (*u x* $*s$ *x*) **apply** (*auto*, *rule right-inverse*) **using** *x-in-cols* **unfolding** *columns-def* **by** *auto*
also have ... = *u x* $*s$ *x* **by** *auto*
also have ... = *u x* $*_R$ *x* **unfolding** *scalar-mult-eq-scaleR* **..**
finally show ($\sum$ *i* | *x* = *column i A*. (*if column i A* $\in$ *S then inverse* (*real* (*card* {*a*. *column i A* = *column a A*})) $*$ *u* (*column i A*) *else 0*) $*s$ *column i A*) = *u x* $*_R$ *x* **.**
qed
also have ... = *setsum* (λ*y*. *?f y* $*_R$ *y*) (*S* $\cup$ ((*columns A*) $-$ *S*)) **apply** (*rule setsum-cong*) **using** *S-in-cols* **by** *auto*
also have ... = *setsum* (λ*y*. *?f y* $*_R$ *y*) *S* + *setsum* (λ*y*. *?f y* $*_R$ *y*) ((*columns A*) $-$ *S*) **apply** (*rule setsum-Un-disjoint*) **using** *S-in-cols finite-S finite-columns* **by** *auto*

also have ... = *setsum* (λy. *?f y* $*_R$ *y*) *S* **by** *simp*
finally show ($\sum i \in UNIV$. (*if column i A* $\in$ *S then inverse* (*real* (*card* {*a*. *column i A* = *column a A*})) * *u* (*column i A*) *else 0*) *s *column i A*) = ($\sum v \in S$. *u v* $*_R$ *v*) **by** *auto*
qed
qed
qed

lemma *matrix-vector-zero*: *A* *v *0* = *0*
unfolding *matrix-vector-mult-def* **by** (*simp add*: *zero-vec-def*)

lemma *vector-matrix-zero*: *0 v** *A* = *0*
unfolding *vector-matrix-mult-def* **by** (*simp add*: *zero-vec-def*)

lemma *vector-matrix-zero′*: *x v** *0* = *0*
unfolding *vector-matrix-mult-def* **by** (*simp add*: *zero-vec-def*)

lemma *transpose-vector*: *x v** *A* = *transpose A* *v *x*
by (*unfold matrix-vector-mult-def vector-matrix-mult-def transpose-def*, *auto*)

lemma *norm-mult-vec*:
fixes *a*::(*real*,′*b*::*finite*) *vec*
shows *norm* (*x* · *x*)=*norm x* * *norm x*
by (*metis inner-real-def norm-cauchy-schwarz-eq norm-mult*)

lemma *norm-equivalence*:
fixes *A*::*real*ˆ′*n*ˆ′*m*
shows ((*transpose A*) *v (*A* *v *x*) = *0*) $\longleftrightarrow$ (*A* *v *x* = *0*)
proof (*auto*)
show *transpose A* *v *0* = *0* **unfolding** *matrix-vector-zero* **..**
next
assume *a*: *transpose A* *v (*A* *v *x*) = *0*
have *eq*: (*x v** (*transpose A*)) = (*A* *v *x*)
by (*metis Cartesian-Euclidean-Space.transpose-transpose transpose-vector*)
have *eq-0*: *0* = (*x v** (*transpose A*)) * (*A* *v *x*)
by (*metis a comm-semiring-1-class.normalizing-semiring-rules*(*7*) *dot-lmul-matrix inner-eq-zero-iff inner-zero-left mult-zero-left transpose-vector*)
hence *0* = *norm* ((*x v** (*transpose A*)) * (*A* *v *x*)) **by** *auto*
also have ... = *norm* ((*A* *v *x*)*(*A* *v *x*)) **unfolding** *eq* **..**
also have ... = *norm* ((*A* *v *x*) · (*A* *v *x*))
by (*metis eq-0 a dot-lmul-matrix eq inner-zero-right norm-zero*)
also have ... = *norm* (*A* *v *x*) ˆ*2* **unfolding** *norm-mult-vec*[*of* (*A* *v *x*)] *power2-eq-square* **..**
finally show *A* *v *x* = *0*
by (*metis* (*lifting*) *field-power-not-zero norm-eq-0-imp*)
qed

lemma *combination-columns*:

fixes *A*::*real*^'*n*^'*m*
assumes *x*: $x \in$ *columns* (*transpose A* ** *A*)
shows $\exists f.$ ($\sum y \in$*columns*(*transpose A*). *f y* $*_R$ *y*) = *x*
proof –
obtain *j* **where** *j*: *x*= *column j* (*transpose A* ** *A*) **using** *x* **unfolding** *columns-def* **by** *auto*
let *?g*=(λ*y*. {*i*. *y*=*column i* (*transpose A*)})
have *inj*: *inj-on ?g* (*columns* (*transpose A*)) **unfolding** *inj-on-def* **unfolding** *columns-def* **by** *auto*
have *union-univ*: $\bigcup$ (*?g*'(*columns* (*transpose A*))) = *UNIV* **unfolding** *columns-def* **by** *auto*
let *?f*=(λ*y*. *card* {*a*. *y* = *column a* (*transpose A*)} * *A* \$ (*SOME a*. *y* = *column a* (*transpose A*)) \$ *j*)
have *column j* (*transpose A* ** *A*) = ($\sum i \in$*UNIV*. (*A*\$*i*\$*j*) $*_R$ (*column i* (*transpose A*)))
unfolding *matrix-matrix-mult-def* **unfolding** *column-def* **by** (*vector*, *auto*, *rule setsum-cong2*, *auto*)
also have ... = *setsum* (λ*i*. (*A*\$*i*\$*j*) $*_R$ (*column i* (*transpose A*))) ($\bigcup$ (*?g*'(*columns* (*transpose A*)))) **unfolding** *union-univ* **..**
also have ... = *setsum* (*setsum* (λ*i*. (*A*\$*i*\$*j*) $*_R$ (*column i* (*transpose A*)))) (*?g*'(*columns* (*transpose A*))) **by** (*rule setsum-Union-disjoint*, *auto*)
also have ... = *setsum* ((*setsum* (λ*i*. (*A*\$*i*\$*j*) $*_R$ (*column i* (*transpose A*)))) $\circ$ *?g*) (*columns* (*transpose A*)) **apply** (*rule setsum-reindex*) **using** *inj* **by** *auto*
also have ... = *setsum* (λ*y*. *?f y* $*_R$ *y*) (*columns* (*transpose A*))
proof (*rule setsum-cong2*, *unfold o-def*)
fix *x* **assume** *x*: $x \in$ *columns* (*transpose A*)
obtain *b* **where** *xb*: *x* = *column b* (*transpose A*) **using** *x* **unfolding** *columns-def* **by** *auto*
have $\forall$ *a c*. *a*$\in$ {*i*. *x* = *column i* (*transpose A*)} $\wedge$ *c*$\in$ {*i*. *x* = *column i* (*transpose A*)} $\longrightarrow$ *A* \$ *a* \$ *j* = *A* \$ *c* \$ *j*
by (*unfold column-def transpose-def*, *auto*, *metis vec-lambda-inverse vec-nth*)
hence *rw-b*: $\forall$ *a*. *a*$\in$ {*i*. *x* = *column i* (*transpose A*)} $\longrightarrow$ *A* \$ *a* \$ *j* = *A* \$ *b* \$ *j* **using** *xb* **by** *fast*
have *Abj*: *A* \$ *b* \$ *j* = *A* \$ (*SOME a*. *x* = *column a* (*Cartesian-Euclidean-Space.transpose A*)) \$ *j*
by (*metis* (*lifting*, *mono-tags*) *xb mem-Collect-eq rw-b some-eq-ex*)
have ($\sum$ *i* | *x* = *column i* (*Cartesian-Euclidean-Space.transpose A*). *A* \$ *i* \$ *j* $*_R$ *column i* (*Cartesian-Euclidean-Space.transpose A*))
= *setsum* (λ*i*. *A* \$ *i* \$ *j* $*_R$ *x*) {*i*. *x* = *column i* (*transpose A*)} **by** *auto*
also have ... = ($\sum i \in$ {*i*. *x* = *column i* (*transpose A*)}. *A* \$ *b* \$ *j* $*_R$ *x*) **using** *rw-b* **by** *auto*
also have ... = *of-nat* (*card* {*i*. *x* = *column i* (*transpose A*)}) * (*A* \$ *b* \$ *j* $*_R$ *x*) **unfolding** *setsum-constant* **by** *auto*
also have ... = (*real* (*card* {*i*. *x* = *column i* (*transpose A*)})) $*_R$ (*A* \$ *b* \$ *j* $*_R$ *x*)
by (*metis* (*no-types*) *real-of-nat-def setsum-constant setsum-constant-scaleR*)
also have ... = (*real* (*card* {*a*. *x* = *column a* (*transpose A*)}) * *A* \$ *b* \$ *j*) $*_R$ *x* **by** *auto*
also have ... = (*real* (*card* {*a*. *x* = *column a* (*transpose A*)}) * *A* \$ (*SOME a*.

x = *column a* (*Cartesian-Euclidean-Space.transpose A*)) $ *j*) $*_R$ *x*
unfolding *Abj* **..**
finally show ($\sum$ *i* | *x* = *column i* (*Cartesian-Euclidean-Space.transpose A*). *A* $ *i* $ *j* $*_R$ *column i* (*Cartesian-Euclidean-Space.transpose A*)) =
(*real* (*card* {*a*. *x* = *column a* (*Cartesian-Euclidean-Space.transpose A*)}) * *A* $ (*SOME a*. *x* = *column a* (*Cartesian-Euclidean-Space.transpose A*)) $ *j*) $*_R$ *x* **.**

qed
finally show *?thesis* **unfolding** *j* **by** *auto*
qed

lemma *rrk-crk*:
fixes *A*::*real^'n^'m*
shows *col-rank A* $\leq$ *row-rank A*
proof −
have *null-space-eq*: *null-space A* = *null-space* (*transpose A* ** *A*) **using** *norm-equivalence* **unfolding** *matrix-vector-mul-assoc* **unfolding** *null-space-def* **by** *fastforce*
have *col-rank-transpose*: *col-rank A* = *col-rank* (*transpose A* ** *A*)
using *rank-nullity-theorem-matrices*[*of A*] *rank-nullity-theorem-matrices*[*of* (*transpose A* ** *A*)]
unfolding *col-space-eq*[*symmetric*]
unfolding *null-space-eq* **by** (*metis col-rank-def nat-add-left-cancel*)
have *col-rank* (*transpose A* ** *A*) $\leq$ *col-rank* (*transpose A*)
proof (*unfold col-rank-def*, *rule subset-le-dim*, *unfold col-space-def*, *unfold span-span*, *clarify*)
fix *x* **assume** *x-in*: *x* $\in$ *span* (*columns* (*transpose A* ** *A*))
show *x* $\in$ *span* (*columns* (*transpose A*))
proof (*rule span-induct*)
show *x* $\in$ *span* (*columns* (*transpose A* ** *A*)) **using** *x-in* **.**
show *subspace* (*span* (*columns* (*transpose A*))) **using** *subspace-span* **.**
fix *y* **assume** *y*: *y* $\in$ *columns* (*transpose A* ** *A*)
show *y* $\in$ *span* (*columns* (*transpose A*))
proof (*unfold span-explicit*, *simp*, *rule exI*[*of - columns* (*transpose A*)], *auto intro*: *combination-columns*[*OF y*])
show *finite* (*columns* (*transpose A*)) **unfolding** *columns-def* **using** *finite-Atleast-Atmost-nat* **by** *auto*
qed
qed
qed
thus *?thesis*
unfolding *col-rank-transpose*[*symmetric*]
unfolding *col-rank-def col-space-def columns-transpose row-rank-def row-space-def*
.
qed

corollary *row-rank-eq-col-rank*:
fixes *A*::*real^'n^'m*
shows *row-rank A* = *col-rank A*

using *rrk-crk*[*of A*] **using** *rrk-crk*[*of transpose A*]
unfolding *col-rank-def row-rank-def row-space-def col-space-def*
unfolding *rows-transpose columns-transpose* **by** *simp*

theorem *rank-col-rank*:
shows $rank\ A = col\text{-}rank\ A$ **unfolding** *rank-def row-rank-eq-col-rank* **..**

theorem *rank-eq-dim-image*:
$rank\ A = dim\ (range\ (\lambda x.\ A *v\ (x{::}real\hat{\ }'n)))$
by (*metis Dim-Formula.col-space-def col-rank-def col-space-eq rank-col-rank*)

theorem *rank-eq-dim-col-space*:
$rank\ A = dim\ (col\text{-}space\ A)$ **using** *rank-col-rank* **unfolding** *col-rank-def* **.**

Following definition and lemmas are obtained from AFP: http://afp.sourceforge.net/browser_info/current/HOL/Tarskis_Geometry/Linear_Algebra2.html

definition
$is\text{-}basis :: (real\hat{\ }('n{::}finite))\ set => bool$ **where**
$is\text{-}basis\ S \equiv independent\ S \wedge span\ S = UNIV$

lemma *card-finite*:
assumes $card\ S = CARD('n{::}finite)$
shows *finite S*
proof −
from ‹$card\ S = CARD('n)$› **have** $card\ S \neq 0$ **by** *simp*
with *card-eq-0-iff* [*of S*] **show** *finite S* **by** *simp*
qed

lemma *independent-is-basis*:
fixes $B :: (real\hat{\ }('n{::}finite))\ set$
shows $independent\ B \wedge card\ B = CARD('n) \longleftrightarrow is\text{-}basis\ B$
proof
assume $independent\ B \wedge card\ B = CARD('n)$
hence *independent B* **and** $card\ B = CARD('n)$ **by** *simp+*
from *card-finite* [*of B*, **where** $'n = 'n$] **and** ‹$card\ B = CARD('n)$›
have *finite B* **by** *simp*
from ‹$card\ B = CARD('n)$›
have $card\ B = dim\ (UNIV :: ((real\hat{\ }'n)\ set))$
by (*simp add*: *dim-UNIV*)
with *card-eq-dim* [*of B UNIV*] **and** ‹*finite B*› **and** ‹*independent B*›
have $span\ B = UNIV$ **by** *auto*
with ‹*independent B*› **show** *is-basis B* **unfolding** *is-basis-def* **..**
next
assume *is-basis B*
hence *independent B* **unfolding** *is-basis-def* **..**
moreover have $card\ B = CARD('n)$
proof −
have $B \subseteq UNIV$ **by** *simp*
moreover

{ **from** ‹*is-basis B*› **have** *UNIV* ⊆ *span B* **and** *independent B*
unfolding *is-basis-def*
by *simp*+ }
ultimately have *card B* = *dim* (*UNIV*::((*real*^'*n*) *set*))
using *basis-card-eq-dim* [*of B UNIV*]
by *simp*
then show *card B* = *CARD*('*n*) **by** (*simp add*: *dim-UNIV*)
qed
ultimately show *independent B* ∧ *card B* = *CARD*('*n*) **..**
qed

lemma *basis-finite*:
fixes *B* :: (*real*^('*n*::*finite*)) *set*
assumes *is-basis B*
shows *finite B*
proof −
from *independent-is-basis* [*of B*] **and** ‹*is-basis B*› **have** *card B* = *CARD*('*n*)
by *simp*
with *card-finite* [*of B*, **where** '*n* = '*n*] **show** *finite B* **by** *simp*
qed

Here ends the facts obtained from AFP: http://afp.sourceforge.net/browser_info/current/HOL/Tarskis_Geometry/Linear_Algebra2.html

Invertible linear maps

We could dispense the property *linear g* using ⟦*linear ?f*; *?g* ∘ *?f* = *id*⟧ ⟹ *linear ?g*

definition *invertible-lf* ::('*a*::*euclidean-space* => '*a*::*euclidean-space*) ⇒ *bool*
where *invertible-lf f* = (*linear f* ∧ (∃ *g*. *linear g* ∧ (*g* ∘ *f* = *id*) ∧ (*f* ∘ *g* = *id*)))

lemma *invertible-lf-intro*[*intro*]:
assumes *linear f* **and** (*g* ∘ *f* = *id*) **and** (*f* ∘ *g* = *id*)
shows *invertible-lf f*
by (*metis assms*(*1*) *assms*(*2*) *invertible-lf-def left-inverse-linear linear-inverse-left*)

lemma *invertible-imp-bijective*:
assumes *invertible-lf f*
shows *bij f*
by (*metis assms bij-betw-comp-iff bij-betw-imp-surj invertible-lf-def inj-on-imageI2 inj-on-imp-bij-betw inv-id surj-id surj-imp-inj-inv*)

lemma *invertible-matrix-imp-invertible-lf*:
fixes *A*::*real*^'*n*^'*n*
assumes *invertible-A*: *invertible A*
shows *invertible-lf* (λ*x*. *A* ∗*v* *x*)
proof −
obtain *B* **where** *AB*: *A*∗∗*B*=*mat 1* **and** *BA*: *B*∗∗*A*=*mat 1* **using** *invertible-A*
unfolding *invertible-def* **by** *blast*

show *?thesis*
proof (*rule invertible-lf-intro* [*of* - ($\lambda x.$ B $*v$ x)])
show *l*: *linear* (*op* $*v$ A) **using** *matrix-vector-mul-linear* **.**
show *id1*: *op* $*v$ $B \circ$ *op* $*v$ $A = id$ **by** (*metis* (*lifting*) *AB BA isomorphism-expand matrix-vector-mul-assoc matrix-vector-mul-lid*)
show *op* $*v$ $A \circ$ *op* $*v$ $B = id$ **by** (*metis l id1 left-inverse-linear linear-inverse-left*)

qed
qed

lemma *invertible-lf-imp-invertible-matrix*:
fixes *f*::*real^'n⇒real^'n*
assumes *invertible-f*: *invertible-lf f*
shows *invertible* (*matrix f*)
proof −
obtain *g* **where** *linear-g*: *linear g* **and** *gf*: ($g \circ f = id$) **and** *fg*: ($f \circ g = id$) **using** *invertible-f* **unfolding** *invertible-lf-def* **by** *auto*
show *?thesis* **proof** (*unfold invertible-def*, *rule exI*[*of* - *matrix g*], *rule conjI*)
show *matrix f* $**$ *matrix g* = *mat 1*
by (*metis* (*no-types*) *fg id-def left-inverse-linear linear-g linear-id matrix-compose matrix-eq matrix-mul-rid matrix-vector-mul matrix-vector-mul-assoc*)
show *matrix g* $**$ *matrix f* = *mat 1*
by (*metis* ‹*matrix f* $**$ *matrix g* = *mat 1*› *matrix-left-right-inverse*)
qed
qed

lemma *invertible-matrix-iff-invertible-lf*:
fixes *A*::*real^'n^'n*
shows *invertible A* ⟷ *invertible-lf* ($\lambda x.$ A $*v$ x)
by (*metis invertible-lf-imp-invertible-matrix invertible-matrix-imp-invertible-lf matrix-of-matrix-vector-mul*)

lemma *invertible-matrix-iff-invertible-lf'*:
fixes *f*::*real^'n⇒real^'n*
assumes *linear-f*: *linear f*
shows *invertible* (*matrix f*) ⟷ *invertible-lf f*
by (*metis* (*lifting*) *assms invertible-matrix-iff-invertible-lf matrix-vector-mul*)

lemma *invertible-matrix-mult-right-rank*:
fixes *A*::*real^'n^'m* **and** *Q*::*real^'n^'n*
assumes *invertible-Q*: *invertible Q*
shows *rank* ($A**Q$) = *rank A*
proof −
def *TQ*==($\lambda x.$ Q $*v$ x)
def *TA*≡($\lambda x.$ A $*v$ x)
def *TAQ*==($\lambda x.$ ($A**Q$) $*v$ x)
have *invertible-lf TQ* **using** *invertible-matrix-imp-invertible-lf*[*OF invertible-Q*] **unfolding** *TQ-def* **.**
hence *bij-TQ*: *bij TQ* **using** *invertible-imp-bijective* **by** *auto*

have *range TAQ* = *range* (*TA* ∘ *TQ*) **unfolding** *TQ-def TA-def TAQ-def o-def matrix-vector-mul-assoc* **..**
also have ... = *TA* ‘(*range TQ*) **unfolding** *image-compose* **..**
also have ... = *TA* ‘ (*UNIV*) **using** *bij-is-surj*[*OF bij-TQ*] **by** *simp*
finally have *range TAQ* = *range TA* **.**
thus *?thesis* **unfolding** *rank-eq-dim-image TAQ-def TA-def* **by** *simp*
qed

lemma *subspace-image-invertible-mat*:
fixes *P*::*real*^'*m*^'*m*
assumes *inv-P*: *invertible P*
and *sub-W*: *subspace W*
shows *subspace* ((λ*x*. *P* ∗*v x*)‘ *W*)
by (*metis* (*lifting*) *matrix-vector-mul-linear sub-W subspace-linear-image*)

lemma *dim-image-invertible-mat*:
fixes *P*::*real*^'*m*^'*m*
assumes *inv-P*: *invertible P*
and *sub-W*: *subspace W*
shows *dim* ((λ*x*. *P* ∗*v x*)‘ *W*) = *dim W*
proof −
obtain *B* **where** *B-in-W*: *B*⊆*W* **and** *ind-B*: *independent B* **and** *W-in-span-B*: *W* ⊆ *span B* **and** *card-B-eq-dim-W*: *card B* = *dim W*
using *basis-exists* **by** *blast*
def *L*≡(λ*x*. *P* ∗*v x*)
def *C*≡*L*‘*B*
have *finite-B*: *finite B* **using** *indep-card-eq-dim-span*[*OF ind-B*] **by** *simp*
have *linear-L*: *linear L* **using** *matrix-vector-mul-linear* **unfolding** *L-def* **.**
have *finite-C*: *finite C* **using** *indep-card-eq-dim-span*[*OF ind-B*] **unfolding** *C-def* **by** *simp*
have *inv-TP*: *invertible-lf* (λ*x*. *P* ∗*v x*) **using** *invertible-matrix-imp-invertible-lf*[*OF inv-P*] **.**
have *inj-on-LW*: *inj-on L W* **using** *invertible-imp-bijective*[*OF inv-TP*] **unfolding** *bij-def L-def* **unfolding** *inj-on-def*
by *blast*
hence *inj-on-LB*: *inj-on L B* **unfolding** *inj-on-def* **using** *B-in-W* **by** *auto*
have *ind-D*: *independent C*
proof (*rule independent-if-scalars-zero*[*OF finite-C*], *clarify*)
fix *f x*
assume *setsum*: ($\sum$ *x*∈*C*. *f x* $*_R$ *x*) = *0* **and** *x*: *x* ∈ *C*
obtain *y* **where** *Ly-eq-x*: *L y* = *x* **and** *y*: *y* ∈ *B* **using** *x* **unfolding** *C-def L-def* **by** *auto*
have ($\sum$ *x*∈*C*. *f x* $*_R$ *x*) = *setsum* ((λ*x*. *f x* $*_R$ *x*) ∘ *L*) *B* **unfolding** *C-def* **by** (*rule setsum-reindex*[*OF inj-on-LB*])
also have ... = *setsum* (λ*x*. *f* (*L x*) $*_R$ *L x*) *B* **unfolding** *o-def* **..**
also have ... = *setsum* (λ*x*. ((*f* ∘ *L*) *x*) $*_R$ *L x*) *B* **using** *o-def* **by** *auto*

also have ... = L (*setsum* ($\lambda x.$ (($f \circ L$) x) $*_R$ x) B) **by** (*rule linear-setsum-mul*[*OF linear-L finite-B*,*symmetric*])
finally have *rw*: ($\sum x{\in}C.\ f\ x *_R x$) = L ($\sum x{\in}B.$ ($f \circ L$) $x *_R x$) **.**
have ($\sum x{\in}B.$ ($f \circ L$) $x *_R x$) $\in W$ **by** (*rule subspace-setsum*[*OF sub-W finite-B*], *auto simp add*: *B-in-W set-rev-mp sub-W subspace-mul*)
hence ($\sum x{\in}B.$ ($f \circ L$) $x *_R x$)=*0* **using** *setsum rw* **using** *linear-injective-on-subspace-0*[*OF linear-L sub-W*] **using** *inj-on-LW* **by** *auto*
hence ($f \circ L$) y = *0* **using** *scalars-zero-if-independent*[*OF finite-B ind-B*, *of* ($f \circ L$)] **using** *y* **by** *auto*
thus $f\ x$ = *0* **unfolding** *o-def Ly-eq-x* **.**
qed
have L‘ $W \subseteq$ *span C*
proof (*unfold span-finite*[*OF finite-C*], *clarify*)
fix *xa* **assume** *xa-in-W*: *xa* $\in W$
obtain *g* **where** *setsum-g*: *setsum* ($\lambda x.\ g\ x *_R x$) B = *xa* **using** *span-finite*[*OF finite-B*] *W-in-span-B xa-in-W* **by** *blast*
show $\exists\, u.$ ($\sum v{\in}C.\ u\ v *_R v$) = L *xa*
proof (*rule exI*[*of* - $\lambda x.\ g$ (*THE* $y.\ y \in B \wedge x{=}L\ y$)])
have L *xa* = L (*setsum* ($\lambda x.\ g\ x *_R x$) B) **using** *setsum-g* **by** *simp*
also have ... = *setsum* ($\lambda x.\ g\ x *_R L\ x$) B **using** *linear-setsum-mul*[*OF linear-L finite-B*] **.**
also have ... = *setsum* ($\lambda x.\ g$ (*THE* $y.\ y \in B \wedge x{=}L\ y$) $*_R x$) (L‘B)
proof (*unfold setsum-reindex*[*OF inj-on-LB*], *unfold o-def*, *rule setsum-cong2*)
fix *x* **assume** *x-in-B*: $x{\in}B$
have *x-eq-the*:x = (*THE* $y.\ y \in B \wedge L\ x = L\ y$)
proof (*rule the-equality*[*symmetric*])
show $x \in B \wedge L\ x = L\ x$ **using** *x-in-B* **by** *auto*
show $\bigwedge y.\ y \in B \wedge L\ x = L\ y \Longrightarrow y = x$ **using** *inj-on-LB x-in-B* **unfolding** *inj-on-def* **by** *fast*
qed
show $g\ x *_R L\ x = g$ (*THE* $y.\ y \in B \wedge L\ x = L\ y$) $*_R L\ x$ **using** *x-eq-the* **by** *simp*
qed
finally show ($\sum v{\in}C.\ g$ (*THE* $y.\ y \in B \wedge v = L\ y$) $*_R v$) = L *xa* **unfolding** *C-def* **..**
qed
qed
have *card C* = *card B* **using** *card-image*[*OF inj-on-LB*] **unfolding** *C-def* **.**
thus *?thesis*
by (*metis Convex-Euclidean-Space.span-eq L-def dim-image-eq inj-on-LW linear-L sub-W*)
qed

lemma *invertible-matrix-mult-left-rank*:
fixes *A*::*real*^′*n*^′*m* **and** *P*::*real*^′*m*^′*m*
assumes *invertible-P*: *invertible P*
shows *rank* ($P{*}{*}A$) = *rank A*
proof –

def *TP*==($\lambda x.$ *P* $*v$ *x*)
def *TA*≡($\lambda x.$ *A* $*v$ *x*)
def *TPA*==($\lambda x.$ (*P*$**$*A*) $*v$ *x*)
have *sub*: *subspace* (*range* (*op* $*v$ *A*)) **by** (*metis matrix-vector-mul-linear subspace-UNIV subspace-linear-image*)
have *dim* (*range TPA*) = *dim* (*range* (*TP* ∘ *TA*)) **unfolding** *TP-def TA-def TPA-def o-def matrix-vector-mul-assoc* **..**
also have ... = *dim* (*range TA*) **using** *dim-image-invertible-mat*[*OF invertible-P sub*] **unfolding** *TP-def TA-def o-def image-compose*[*symmetric*] **.**
finally show *?thesis* **unfolding** *rank-eq-dim-image TPA-def TA-def* **.**
qed

corollary *invertible-matrices-mult-rank*:
fixes *A*::*real*^'*n*^'*m* **and** *P*::*real*^'*m*^'*m* **and** *Q*::*real*^'*n*^'*n*
assumes *invertible-P*: *invertible P*
and *invertible-Q*: *invertible Q*
shows *rank* (*P*$**$*A*$**$*Q*) = *rank A*
using *invertible-matrix-mult-right-rank*[*OF invertible-Q*] **using** *invertible-matrix-mult-left-rank*[*OF invertible-P*] **by** *metis*

lemma *invertible-matrix-mult-left-rank'*:
fixes *A*::*real*^'*n*^'*m* **and** *P*::*real*^'*m*^'*m*
assumes *invertible-P*: *invertible P* **and** *B-eq-PA*: *B*=*P*$**$*A*
shows *rank B* = *rank A*
proof −
have *rank B* = *rank* (*P*$**$*A*) **using** *B-eq-PA* **by** *auto*
also have ... = *rank A* **using** *invertible-matrix-mult-left-rank*[*OF invertible-P*] **by** *auto*
finally show *?thesis* **.**
qed

lemma *invertible-matrix-mult-right-rank'*:
fixes *A*::*real*^'*n*^'*m* **and** *Q*::*real*^'*n*^'*n*
assumes *invertible-Q*: *invertible Q* **and** *B-eq-PA*: *B*=*A*$**$*Q*
shows *rank B* = *rank A* **by** (*metis B-eq-PA invertible-Q invertible-matrix-mult-right-rank*)

lemma *invertible-matrices-rank'*:
fixes *A*::*real*^'*n*^'*m* **and** *P*::*real*^'*m*^'*m* **and** *Q*::*real*^'*n*^'*n*
assumes *invertible-P*: *invertible P* **and** *invertible-Q*: *invertible Q* **and** *B-eq-PA*: *B*=*P*$**$*A*$**$*Q*
shows *rank B* = *rank A* **by** (*metis B-eq-PA invertible-P invertible-Q invertible-matrices-mult-rank*)

Some definitions:

definition *set-of-vector* :: '*a*^'*n* ⇒ '*a set*
where *set-of-vector A* = {*A*$*i* |*i*. *i* ∈ *UNIV*}

lemma *basis-image-linear*:
assumes *invertible-lf*: *invertible-lf f*

and *basis-X*: *is-basis* (*set-of-vector X*)
shows *is-basis* (*f*‘ (*set-of-vector X*))
proof (*rule iffD1*[*OF independent-is-basis*], *rule conjI*)
have *card* (*f* ‘ *set-of-vector X*) = *card* (*set-of-vector X*)
by (*rule card-image*[*of f set-of-vector X*], *metis invertible-imp-bijective*[*OF invertible-lf*] *bij-def inj-eq inj-on-def*)
also have ... = *card* (*UNIV*::'*a set*) **using** *independent-is-basis basis-X* **by** *auto*
finally show *card* (*f* ‘ *set-of-vector X*) = *card* (*UNIV*::'*a set*) **.**
show *independent* (*f* ‘ *set-of-vector X*)
proof (*rule independent-injective-image*)
show *independent* (*set-of-vector X*) **using** *basis-X* **unfolding** *is-basis-def* **by** *simp*
show *linear f* **using** *invertible-lf* **unfolding** *invertible-lf-def* **by** *simp*
show *inj f* **using** *invertible-imp-bijective*[*OF invertible-lf*] **unfolding** *bij-def* **by** *simp*
qed
qed

definition *cart-basis'* :: *real*^'*n*^'*n*
where *cart-basis'* = (χ *i*. *axis i 1*)

Properties about *cart-basis'* = (χ*i*. *axis i 1*)

lemma *set-of-vector-cart-basis'*:
shows (*set-of-vector cart-basis'*) = {*axis i 1* :: *real*^'*n* | *i*. *i* ∈ (*UNIV* :: '*n set*)}
unfolding *set-of-vector-def cart-basis'-def* **by** *auto*

lemma *cart-basis'-i*: *cart-basis'* $ *i* = *axis i 1* **unfolding** *cart-basis'-def* **by** *simp*

lemma *finite-cart-basis'*:
shows *finite* (*set-of-vector cart-basis'*)
proof−
def *f*==λ*i*. (*cart-basis'*::*real*^'*a*^'*a*) $ *i*
show *?thesis* **unfolding** *set-of-vector-def* **using** *finite-Atleast-Atmost-nat*[*of f*] **unfolding** *f-def* **.**
qed

lemma *axis-Basis*:{*axis i* (*1*::*real*) |*i*. *i* ∈ (*UNIV*::('*a*::*finite set*))} = *Basis*
proof (*auto*)
fix *i*::'*n*::*finite* **show** *axis i* (*1*::*real*) ∈ *Basis* **proof** (*rule axis-in-Basis*)
show (*1*::*real*) ∈ *Basis* **using** *Basis-real-def* **by** *simp*
qed
next
fix *x*::*real*^'*n* **assume** *x*: *x* ∈ *Basis* **show** ∃ *i*. *x* = *axis i 1* **using** *x* **unfolding** *Basis-vec-def* **by** *auto*
qed

lemma *span-stdbasis*:*span* {*axis i 1* :: *real*^'*n* | *i*. *i* ∈ (*UNIV* :: '*n set*)} = *UNIV*
unfolding *span-Basis*[*symmetric*] **unfolding** *axis-Basis* **by** *auto*

lemma *independent-stdbasis*: *independent* $\{$*axis i 1* ::*real*$\hat{}$$'$*n* $|$*i*. *i*$\in$ (*UNIV* :: $'$*n* *set*)$\}$
  by (*rule independent-substdbasis*, *auto*, *rule axis-in-Basis*, *auto*)

lemma *span-cart-basis*$'$:
  shows *span* (*set-of-vector cart-basis*$'$) = *UNIV*
   unfolding *set-of-vector-def* **unfolding** *cart-basis*$'$*-def* **using** *span-stdbasis* **by** *auto*

lemma *is-basis-cart-basis*$'$: *is-basis* (*set-of-vector* (*cart-basis*$'$))
 by (*metis* (*lifting*) *independent-stdbasis is-basis-def set-of-vector-cart-basis*$'$ *span-stdbasis*)

lemma *basis-expansion-cart-basis*$'$:*setsum* (λ*i*. *x*\$*i* $*_R$ *cart-basis*$'$ \$ *i*) *UNIV* = *x*
unfolding *cart-basis*$'$*-def* **using** *basis-expansion* **apply** *auto*
proof −
assume *ass*:($\bigwedge$*x*::(*real*, $'$*a*) *vec*. ($\sum$*i*$\in$*UNIV*. *x* \$ *i* $*$*s axis i 1*) = *x*)
have ($\sum$*i*$\in$*UNIV*. *x* \$ *i* $*$*s axis i 1*) = *x* **using** *ass*[*of x*] **.**
thus ($\sum$*i*$\in$*UNIV*. *x* \$ *i* $*_R$ *axis i 1*) = *x* **unfolding** *scalar-mult-eq-scaleR* **.**
qed

lemma *basis-expansion-unique*:
  setsum (λ*i*. *f i* $*$*s axis* (*i*::$'$*n*::*finite*) *1*) *UNIV* = (*x*::($'$*a*::*comm-ring-1*) $\hat{}$$'$*n*) <−> ($\forall$*i*. *f i* = *x*\$*i*)
proof (*auto simp add*: *basis-expansion*)
  fix *i*::$'$*n*
  have *univ-rw*: *UNIV* = (*UNIV* − $\{$*i*$\}$) $\cup$ $\{$*i*$\}$ **by** *fastforce*
  have ($\sum$*x*$\in$*UNIV*. *f x* $*$ *axis x 1* \$ *i*) = *setsum* (λ*x*. *f x* $*$ *axis x 1* \$ *i*) (*UNIV* − $\{$*i*$\}$ $\cup$ $\{$*i*$\}$) **using** *univ-rw* **by** *simp*
  also have ... = *setsum* (λ*x*. *f x* $*$ *axis x 1* \$ *i*) (*UNIV* − $\{$*i*$\}$) + *setsum* (λ*x*. *f x* $*$ *axis x 1* \$ *i*) $\{$*i*$\}$ **by** (*rule setsum-Un-disjoint*, *auto*)
  also have ... = *f i* **unfolding** *axis-def* **by** *auto*
  finally show *f i* = ($\sum$*x*$\in$*UNIV*. *f x* $*$ *axis x 1* \$ *i*) **..**
qed

lemma *basis-expansion-cart-basis*$'$*-unique*: *setsum* (λ*i*. *f* (*cart-basis*$'$ \$ *i*) $*_R$ *cart-basis*$'$ \$ *i*) *UNIV* = *x* <−> ($\forall$*i*. *f* (*cart-basis*$'$ \$ *i*) = *x*\$*i*)
  using *basis-expansion-unique* **unfolding** *cart-basis*$'$*-def*
  by (*simp add*: *vec-eq-iff setsum-delta if-distrib cong del*: *if-weak-cong*)

lemma *basis-expansion-cart-basis*$'$*-unique*$'$: *setsum* (λ*i*. *f i* $*_R$ *cart-basis*$'$ \$ *i*) *UNIV* = *x* <−> ($\forall$*i*. *f i* = *x*\$*i*)
  using *basis-expansion-unique* **unfolding** *cart-basis*$'$*-def*
  by (*simp add*: *vec-eq-iff setsum-delta if-distrib cong del*: *if-weak-cong*)

Properties of *is-basis ?S* $\equiv$ *independent ?S* $\wedge$ *span ?S* = *UNIV*

lemma *setsum-basis-eq*:
  fixes *X*::*real*$\hat{}$$'$*n*$\hat{}$$'$*n*
  assumes *is-basis*:*is-basis* (*set-of-vector X*)
  shows *setsum* (λ*x*. *f x* $*_R$ *x*) (*set-of-vector X*) = *setsum* (λ*i*. *f* (*X*\$*i*) $*_R$ (*X*\$*i*))

UNIV
proof (*rule setsum-reindex-cong*[*of* $\lambda i.$ *X*\$*i*])
  show *fact-1*: *set-of-vector X* = *range* (*op* \$ *X*) **unfolding** *set-of-vector-def* **by** *auto*
  have *card-set-of-vector*:*card*(*set-of-vector X*) = *CARD*('*n*) **using** *independent-is-basis*[*of set-of-vector X*] **using** *is-basis* **by** *auto*
  show *inj* (*op* \$ *X*)
  proof (*rule eq-card-imp-inj-on*)
    show *finite* (*UNIV*::'*n set*) **using** *finite-class.finite-UNIV* **.**
    show *card* (*range* (*op* \$ *X*)) = *card* (*UNIV*::'*n set*) **using** *card-set-of-vector* **using** *fact-1* **unfolding** *set-of-vector-def* **by** *simp*
  qed
  show $\bigwedge a.\ a \in UNIV \Longrightarrow f\ (X\ \$\ a) *_R X\ \$\ a = f\ (X\ \$\ a) *_R X\ \$\ a$ **by** *simp*
qed

corollary *setsum-basis-eq2*:
  fixes *X*::*real*^'*n*^'*n*
  assumes *is-basis*:*is-basis* (*set-of-vector X*)
  shows *setsum* ($\lambda x.\ f\,x *_R x$) (*set-of-vector X*) = *setsum* ($\lambda i.$ ($f \circ op$ \$ *X*) $i *_R$ (*X*\$*i*)) *UNIV* **using** *setsum-basis-eq*[*OF is-basis*] **by** *simp*

lemma *inj-op-nth*:
  fixes *X*::*real*^'*n*^'*n*
  assumes *is-basis*: *is-basis* (*set-of-vector X*)
  shows *inj* (*op* \$ *X*)
proof −
  have *fact-1*: *set-of-vector X* = *range* (*op* \$ *X*) **unfolding** *set-of-vector-def* **by** *auto*
  have *card-set-of-vector*:*card*(*set-of-vector X*) = *CARD*('*n*) **using** *independent-is-basis*[*of set-of-vector X*] **using** *is-basis* **by** *auto*
  show *inj* (*op* \$ *X*)
  proof (*rule eq-card-imp-inj-on*)
    show *finite* (*UNIV*::'*n set*) **using** *finite-class.finite-UNIV* **.**
    show *card* (*range* (*op* \$ *X*)) = *card* (*UNIV*::'*n set*) **using** *card-set-of-vector* **using** *fact-1* **unfolding** *set-of-vector-def* **by** *simp*
  qed
qed

lemma *basis-UNIV*:
  fixes *X*::*real*^'*n*^'*n*
  assumes *is-basis*: *is-basis* (*set-of-vector X*)
  shows $UNIV = \{x.\ \exists g.\ (\sum i{\in}UNIV.\ g\ i *_R X\$i) = x\}$
proof −
  have $UNIV = \{x.\ \exists g.\ (\sum i{\in}(\textit{set-of-vector}\ X).\ g\ i *_R i) = x\}$ **using** *is-basis* **unfolding** *is-basis-def* **using** *span-finite*[*OF basis-finite*[*OF is-basis*]] **by** *simp*
  also have ... $\subseteq \{x.\ \exists g.\ (\sum i{\in}UNIV.\ g\ i *_R X\$i) = x\}$
  proof (*clarify*)
    fix *f*
    show $\exists g.\ (\sum i{\in}UNIV.\ g\ i *_R X\ \$\ i) = (\sum i{\in}\textit{set-of-vector}\ X.\ f\ i *_R i)$

proof (*rule exI*[*of* - ($\lambda i.$ ($f \circ op$ \$ X) i)], *unfold o-def*, *rule setsum-reindex-cong*[*symmetric*, *of op* \$ X])
show *fact-1*: *set-of-vector* X = *range* (*op* \$ X) **unfolding** *set-of-vector-def* **by** *auto*
have *card-set-of-vector*:*card*(*set-of-vector* X) = *CARD*('*n*) **using** *independent-is-basis*[*of set-of-vector* X] **using** *is-basis* **by** *auto*
show *inj* (*op* \$ X) **using** *inj-op-nth*[*OF is-basis*] **.**
show $\bigwedge a.\ a \in UNIV \Longrightarrow f\ (X\ \$\ a) *_R X\ \$\ a = f\ (X\ \$\ a) *_R X\ \$\ a$ **by** *simp*
qed
qed
finally show *?thesis* **by** *auto*
qed

lemma *scalars-zero-if-basis*:
fixes X::*real*^'*n*^'*n*
assumes *is-basis*: *is-basis* (*set-of-vector* X) **and** *setsum*: ($\sum i{\in}(UNIV{::}'n\ set).\ f\ i *_R X\$i) = 0$)
shows $\forall i{\in}(UNIV{::}'n\ set).\ f\ i = 0$
proof −
have *ind-X*: *independent* (*set-of-vector* X) **using** *is-basis* **unfolding** *is-basis-def* **by** *simp*
have *finite-X*:*finite* (*set-of-vector* X) **using** *basis-finite*[*OF is-basis*] **.**
have *1*: ($\forall g.\ (\sum v{\in}(set\text{-}of\text{-}vector\ X).\ g\ v *_R v) = 0 \longrightarrow (\forall v{\in}(set\text{-}of\text{-}vector\ X).\ g\ v = 0)$) **using** *ind-X* **unfolding** *independent-explicit* **using** *finite-X* **by** *auto*
def $g \equiv \lambda v.\ f\ (THE\ i.\ X\ \$\ i = v)$
have ($\sum v{\in}(set\text{-}of\text{-}vector\ X).\ g\ v *_R v) = 0$
proof −
have ($\sum v{\in}(set\text{-}of\text{-}vector\ X).\ g\ v *_R v) = (\sum i{\in}(UNIV{::}'n\ set).\ f\ i *_R X\$i)$
proof (*rule setsum-reindex-cong*)
show *inj* (*op* \$ X) **using** *inj-op-nth*[*OF is-basis*] **.**
show *set-of-vector* X = *range* (*op* \$ X) **unfolding** *set-of-vector-def* **by** *auto*
show $\bigwedge a.\ a \in (UNIV{::}'n\ set) \Longrightarrow f\ a *_R X\ \$\ a = g\ (X\ \$\ a) *_R X\ \$\ a$
proof (*auto*)
fix a
assume $X\ \$\ a \neq 0$
show $f\ a = g\ (X\ \$\ a)$
unfolding *g-def* **using** *inj-op-nth*[*OF is-basis*]
by (*metis* (*lifting*, *mono-tags*) *injD the-equality*)
qed
qed
thus *?thesis* **unfolding** *setsum* **.**
qed
hence *2*: $\forall v{\in}(set\text{-}of\text{-}vector\ X).\ g\ v = 0$ **using** *1* **by** *auto*
show *?thesis*
proof (*clarify*)
fix a
have $g\ (X\$a) = 0$ **using** *2* **unfolding** *set-of-vector-def* **by** *auto*
thus $f\ a = 0$ **unfolding** *g-def* **using** *inj-op-nth*[*OF is-basis*]

by (*metis* (*lifting*, *mono-tags*) *injD the-equality*)
qed
qed

lemma *basis-combination-unique*:
fixes X::*real*^'*n*^'*n*
assumes *basis-X*: *is-basis* (*set-of-vector X*) **and** *setsum-eq*: $(\sum i{\in}UNIV.\ g\ i *_R X\$i) = (\sum i{\in}UNIV.\ f\ i *_R X\$i)$
shows $f=g$
proof (*rule ccontr*)
assume $f{\neq}g$
from *this* **obtain** x **where** *fx-gx*: $f\ x \neq g\ x$ **by** *fast*
have $0{=}(\sum i{\in}UNIV.\ g\ i *_R X\$i) - (\sum i{\in}UNIV.\ f\ i *_R X\$i)$ **using** *setsum-eq* **by** *simp*
also have ... $= (\sum i{\in}UNIV.\ g\ i *_R X\$i - f\ i *_R X\$i)$ **unfolding** *setsum-subtractf*[*symmetric*]
..
also have ... $= (\sum i{\in}UNIV.\ (g\ i - f\ i) *_R X\$i)$ **by** (*rule setsum-cong2*, *simp add*: *scaleR-diff-left*)
also have ... $= (\sum i{\in}UNIV.\ (g - f)\ i *_R X\$i)$ **by** *simp*
finally have *setsum-eq-1*: $0 = (\sum i{\in}UNIV.\ (g - f)\ i *_R X\$i)$ **by** *simp*
have $\forall\, i{\in}UNIV.\ (g - f)\ i = 0$ **by** (*rule scalars-zero-if-basis*[*OF basis-X setsum-eq-1*[*symmetric*]])
hence $(g - f)\ x = 0$ **by** *simp*
hence $f\ x = g\ x$ **by** *simp*
thus *False* **using** *fx-gx* **by** *contradiction*
qed

Definition and properties of the coordinates of a vector (in terms of a particular ordered basis).

definition *coord* :: *real*^'*n*^'*n*⇒*real*^'*n*⇒*real*^'*n*
where *coord X v* $= (\chi\ i.\ (THE\ f.\ v = setsum\ (\lambda x.\ f\ x *_R X\$x)\ UNIV)\ i)$

coord X v is the coordinates of vector v with respect to the basis X

lemma *bij-coord*:
fixes X::*real*^'*n*^'*n*
assumes *basis-X*: *is-basis* (*set-of-vector X*)
shows *bij* (*coord X*)
proof (*unfold bij-def*, *auto*)
show *inj*: *inj* (*coord X*)
proof (*unfold inj-on-def*, *auto*)
fix $x\ y$ **assume** *coord-eq*: *coord X x* = *coord X y*
obtain f **where** f: $(\sum x{\in}UNIV.\ f\ x *_R X\ \$\ x) = x$ **using** *basis-UNIV*[*OF basis-X*] **by** *blast*
obtain g **where** g: $(\sum x{\in}UNIV.\ g\ x *_R X\ \$\ x) = y$ **using** *basis-UNIV*[*OF basis-X*] **by** *blast*
have *the-f*: $(THE\ f.\ x = (\sum x{\in}UNIV.\ f\ x *_R X\ \$\ x)) = f$
proof (*rule the-equality*)
show $x = (\sum x{\in}UNIV.\ f\ x *_R X\ \$\ x)$ **using** f **by** *simp*
show $\bigwedge fa.\ x = (\sum x{\in}UNIV.\ fa\ x *_R X\ \$\ x) \Longrightarrow fa = f$ **using** *basis-combination-unique*[*OF basis-X*] f **by** *simp*

qed
have *the-g*: $(THE\ g.\ y = (\sum x{\in}UNIV.\ g\ x *_R X\ \$\ x)) = g$
proof (*rule the-equality*)
show $y = (\sum x{\in}UNIV.\ g\ x *_R X\ \$\ x)$ **using** *g* **by** *simp*
show $\bigwedge ga.\ y = (\sum x{\in}UNIV.\ ga\ x *_R X\ \$\ x) \Longrightarrow ga = g$ **using** *basis-combination-unique*[*OF basis-X*] *g* **by** *simp*
qed
have $(THE\ f.\ x = (\sum x{\in}UNIV.\ f\ x *_R X\ \$\ x)) = (THE\ g.\ y = (\sum x{\in}UNIV.\ g\ x *_R X\ \$\ x))$
using *coord-eq* **unfolding** *coord-def*
using *vec-lambda-inject*[*of* $(THE\ f.\ x = (\sum x{\in}UNIV.\ f\ x *_R X\ \$\ x))$ $(THE\ f.\ y = (\sum x{\in}UNIV.\ f\ x *_R X\ \$\ x))$]
by *auto*
hence $f = g$ **unfolding** *the-f the-g* **.**
thus $x{=}y$ **using** *f g* **by** *simp*
qed
next
fix x::(*real*, *'n*) *vec*
show $x \in range\ (coord\ X)$
proof (*unfold image-def*, *auto*, *rule exI*[*of - setsum* $(\lambda i.\ x\$i *_R X\$i)$ *UNIV*], *unfold coord-def*)
def $f{\equiv}\lambda i.\ x\$i$
have *the-f*: $(THE\ f.\ (\sum i{\in}UNIV.\ x\ \$\ i *_R X\ \$\ i) = (\sum x{\in}UNIV.\ f\ x *_R X\ \$\ x)) = f$
proof (*rule the-equality*)
show $(\sum i{\in}UNIV.\ x\ \$\ i *_R X\ \$\ i) = (\sum x{\in}UNIV.\ f\ x *_R X\ \$\ x)$ **unfolding** *f-def* **..**
fix *g* **assume** *setsum-eq*: $(\sum i{\in}UNIV.\ x\ \$\ i *_R X\ \$\ i) = (\sum x{\in}UNIV.\ g\ x *_R X\ \$\ x)$
show $g = f$ **using** *basis-combination-unique*[*OF basis-X*] **using** *setsum-eq* **unfolding** *f-def* **by** *simp*
qed
show $x = vec\text{-}lambda\ (THE\ f.\ (\sum i{\in}UNIV.\ x\ \$\ i *_R X\ \$\ i) = (\sum x{\in}UNIV.\ f\ x *_R X\ \$\ x))$ **unfolding** *the-f* **unfolding** *f-def* **using** *vec-lambda-eta*[*of x*] **by** *simp*
qed
qed

lemma *linear-coord*:
fixes X::*real*^*'n*^*'n*
assumes *basis-X*: *is-basis* (*set-of-vector X*)
shows *linear* (*coord X*)
proof (*unfold linear-def coord-def*, *auto*)
fix *x y*::(*real*, *'n*) *vec*
show $vec\text{-}lambda\ (THE\ f.\ x + y = (\sum x{\in}UNIV.\ f\ x *_R X\ \$\ x)) = vec\text{-}lambda\ (THE\ f.\ x = (\sum x{\in}UNIV.\ f\ x *_R X\ \$\ x)) + vec\text{-}lambda\ (THE\ f.\ y = (\sum x{\in}UNIV.\ f\ x *_R X\ \$\ x))$
proof −

obtain f **where** f: $(\sum a{\in}(UNIV{::}'n\ set).\ f\ a *_R X\ \$\ a) = x + y$ **using** *basis-UNIV*[*OF basis-X*] **by** *blast*
obtain g **where** g: $(\sum x{\in}UNIV.\ g\ x *_R X\ \$\ x) = x$ **using** *basis-UNIV*[*OF basis-X*] **by** *blast*
obtain h **where** h: $(\sum x{\in}UNIV.\ h\ x *_R X\ \$\ x) = y$ **using** *basis-UNIV*[*OF basis-X*] **by** *blast*
def $t{\equiv}\lambda i.\ g\ i + h\ i$
have *the-f*: $(THE\ f.\ x + y = (\sum x{\in}UNIV.\ f\ x *_R X\ \$\ x)) = f$
proof (*rule the-equality*)
show $x + y = (\sum x{\in}UNIV.\ f\ x *_R X\ \$\ x)$ **using** f **by** *simp*
show $\bigwedge fa.\ x + y = (\sum x{\in}UNIV.\ fa\ x *_R X\ \$\ x) \Longrightarrow fa = f$ **using** *basis-combination-unique*[*OF basis-X*] f **by** *simp*
qed
have *the-g*: $(THE\ g.\ x = (\sum x{\in}UNIV.\ g\ x *_R X\ \$\ x)) = g$
proof (*rule the-equality*)
show $x = (\sum x{\in}UNIV.\ g\ x *_R X\ \$\ x)$ **using** g **by** *simp*
show $\bigwedge ga.\ x = (\sum x{\in}UNIV.\ ga\ x *_R X\ \$\ x) \Longrightarrow ga = g$ **using** *basis-combination-unique*[*OF basis-X*] g **by** *simp*
qed
have *the-h*: $(THE\ h.\ y = (\sum x{\in}UNIV.\ h\ x *_R X\ \$\ x)) = h$
proof (*rule the-equality*)
show $y = (\sum x{\in}UNIV.\ h\ x *_R X\ \$\ x)$ **using** h **..**
show $\bigwedge ha.\ y = (\sum x{\in}UNIV.\ ha\ x *_R X\ \$\ x) \Longrightarrow ha = h$ **using** *basis-combination-unique*[*OF basis-X*] h **by** *simp*
qed
have $(\sum a{\in}(UNIV{::}'n\ set).\ f\ a *_R X\ \$\ a) = (\sum x{\in}UNIV.\ g\ x *_R X\ \$\ x) + (\sum x{\in}UNIV.\ h\ x *_R X\ \$\ x)$ **using** $f\ g\ h$ **by** *simp*
also have ... $= (\sum x{\in}UNIV.\ g\ x *_R X\ \$\ x + h\ x *_R X\ \$\ x)$ **unfolding** *setsum-addf*[*symmetric*] **..**
also have ... $= (\sum x{\in}UNIV.\ (g\ x + h\ x) *_R X\ \$\ x)$ **by** (*rule setsum-cong2*, *simp add*: *scaleR-left-distrib*)
also have ... $= (\sum x{\in}UNIV.\ t\ x *_R X\ \$\ x)$ **unfolding** *t-def* **..**
finally have $(\sum a{\in}UNIV.\ f\ a *_R X\ \$\ a) = (\sum x{\in}UNIV.\ t\ x *_R X\ \$\ x)$ **.**
hence $f{=}t$ **using** *basis-combination-unique*[*OF basis-X*] **by** *auto*
thus *?thesis*
by (*unfold the-f the-g the-h*, *vector*, *auto*, *unfold f g h t-def*, *simp*)
qed
next
fix $c\ x$
show *vec-lambda* $(THE\ f.\ c *_R x = (\sum x{\in}UNIV.\ f\ x *_R X\ \$\ x)) = c *_R$ *vec-lambda* $(THE\ f.\ x = (\sum x{\in}UNIV.\ f\ x *_R X\ \$\ x))$
proof −
obtain f **where** f: $(\sum x{\in}UNIV.\ f\ x *_R X\ \$\ x) = c *_R x$ **using** *basis-UNIV*[*OF basis-X*] **by** *blast*
obtain g **where** g: $(\sum x{\in}UNIV.\ g\ x *_R X\ \$\ x) = x$ **using** *basis-UNIV*[*OF basis-X*] **by** *blast*
def $t{\equiv}\lambda i.\ c *_R g\ i$
have *the-f*: $(THE\ f.\ c *_R x = (\sum x{\in}UNIV.\ f\ x *_R X\ \$\ x)) = f$
proof (*rule the-equality*)

show $c *_R x = (\sum x\in UNIV.\ f\ x *_R X \$ x)$ **using** f **..**
show $\bigwedge fa.\ c *_R x = (\sum x\in UNIV.\ fa\ x *_R X \$ x) \Longrightarrow fa = f$ **using** *basis-combination-unique*[*OF basis-X*] f **by** *simp*
qed
have *the-g*: $(THE\ g.\ x = (\sum x\in UNIV.\ g\ x *_R X \$ x)) = g$**proof** (*rule the-equality*)
show $x = (\sum x\in UNIV.\ g\ x *_R X \$ x)$ **using** g **..**
show $\bigwedge ga.\ x = (\sum x\in UNIV.\ ga\ x *_R X \$ x) \Longrightarrow ga = g$ **using** *basis-combination-unique*[*OF basis-X*] g **by** *simp*
qed
have $(\sum x\in UNIV.\ f\ x *_R X \$ x) = c *_R (\sum x\in UNIV.\ g\ x *_R X \$ x)$ **using** $f\ g$ **by** *simp*
also have ... $= (\sum x\in UNIV.\ c *_R g\ x *_R X \$ x)$ **by** (*rule scaleR-setsum-right*)
also have ... $= (\sum x\in UNIV.\ t\ x *_R X \$ x)$ **unfolding** *t-def* **by** *simp*
finally have $(\sum x\in UNIV.\ f\ x *_R X \$ x) = (\sum x\in UNIV.\ t\ x *_R X \$ x)$ **.**
hence $f=t$ **using** *basis-combination-unique*[*OF basis-X*] **by** *auto*
thus *?thesis*
by (*unfold the-f the-g*, *vector*, *auto*, *unfold t-def*, *auto*)
qed
qed

lemma *coord-eq*:
assumes *basis-X*:*is-basis* (*set-of-vector X*)
and *coord-eq*: *coord X v* = *coord X w*
shows $v = w$
proof −
have $\forall i.\ (THE\ f.\ \forall i.\ v \$ i = (\sum x\in UNIV.\ f\ x * X \$ x \$ i))\ i = (THE\ f.\ \forall i.\ w \$ i = (\sum x\in UNIV.\ f\ x * X \$ x \$ i))\ i$ **using** *coord-eq*
unfolding *coord-eq coord-def vec-eq-iff* **by** *simp*
hence *the-eq*: $(THE\ f.\ \forall i.\ v \$ i = (\sum x\in UNIV.\ f\ x * X \$ x \$ i)) = (THE\ f.\ \forall i.\ w \$ i = (\sum x\in UNIV.\ f\ x * X \$ x \$ i))$ **by** *auto*
obtain f **where** f: $(\sum x\in UNIV.\ f\ x *_R X \$ x)= v$ **using** *basis-UNIV*[*OF basis-X*] **by** *blast*
obtain g **where** g: $(\sum x\in UNIV.\ g\ x *_R X \$ x)= w$ **using** *basis-UNIV*[*OF basis-X*] **by** *blast*
have *the-f*: $(THE\ f.\ \forall i.\ v \$ i = (\sum x\in UNIV.\ f\ x * X \$ x \$ i)) = f$
proof (*rule the-equality*)
show $\forall i.\ v \$ i = (\sum x\in UNIV.\ f\ x * X \$ x \$ i)$ **using** f **by** *auto*
fix *fa* **assume** $\forall i.\ v \$ i = (\sum x\in UNIV.\ fa\ x * X \$ x \$ i)$
hence $\forall i.\ v \$ i = (\sum x\in UNIV.\ fa\ x *_R X \$ x) \$ i$ **unfolding** *setsum-component* **by** *simp*
hence *fa*: $v = (\sum x\in UNIV.\ fa\ x *_R X \$ x)$ **unfolding** *vec-eq-iff* **.**
show $fa = f$ **using** *basis-combination-unique*[*OF basis-X*] f *fa* **by** *simp*
qed
have *the-g*: $(THE\ g.\ \forall i.\ w \$ i = (\sum x\in UNIV.\ g\ x * X \$ x \$ i)) = g$
proof (*rule the-equality*)
show $\forall i.\ w \$ i = (\sum x\in UNIV.\ g\ x * X \$ x \$ i)$ **using** g **by** *auto*
fix *fa* **assume** $\forall i.\ w \$ i = (\sum x\in UNIV.\ fa\ x * X \$ x \$ i)$

hence $\forall\, i.\ w\ \$\ i = (\sum x{\in}UNIV.\ fa\ x *_R X\ \$\ x)\ \$\ i$ **unfolding** *setsum-component* **by** *simp*
hence *fa*: $w = (\sum x{\in}UNIV.\ fa\ x *_R X\ \$\ x)$ **unfolding** *vec-eq-iff* **.**
show $fa = g$ **using** *basis-combination-unique*[*OF basis-X*] *g fa* **by** *simp*
qed
have $f{=}g$ **using** *the-eq* **unfolding** *the-f the-g* **.**
thus $v{=}w$ **using** *f g* **by** *blast*
qed

Definitions of matrix of change of basis and matrix of a linear transformation with respect to two bases:

definition *matrix-change-of-basis* :: *real^'n^'n* ⇒*real^'n^'n*⇒*real^'n^'n*
where *matrix-change-of-basis X Y* = (χ *i j*. (*coord Y* (*X*\$*j*)) \$ *i*)

definition *matrix'* :: *real^'n^'n* ⇒ *real^'m^'m* ⇒ (*real^'n* => *real^'m*) ⇒ *real^'n^'m*
where *matrix' X Y f*= (χ *i j*. (*coord Y* (*f*(*X*\$*j*))) \$ *i*)

Properties of *matrix' ?X ?Y ?f* = (χ*i j*. *coord ?Y* (*?f* (*?X* \$ *j*)) \$ *i*)

lemma *matrix'-eq-matrix*:
defines *cart-basis-Rn*: *cart-basis-Rn* == (*cart-basis'*)::*real^'n^'n* **and** *cart-basis-Rm*:*cart-basis-Rm* == (*cart-basis'*)::*real^'m^'m*
assumes *lf*: *linear f*
shows *matrix'* (*cart-basis-Rn*) (*cart-basis-Rm*) *f* = *matrix f*
proof (*unfold matrix-def matrix'-def coord-def*, *vector*, *auto*)
fix *i j*
have *basis-Rn*:*is-basis* (*set-of-vector cart-basis-Rn*) **using** *is-basis-cart-basis'* **unfolding** *cart-basis-Rn* **.**
have *basis-Rm*:*is-basis* (*set-of-vector cart-basis-Rm*) **using** *is-basis-cart-basis'* **unfolding** *cart-basis-Rm* **.**
obtain *g* **where** *setsum-g*: $(\sum x{\in}UNIV.\ g\ x *_R$ (*cart-basis-Rm* \$ *x*)) = *f* (*cart-basis-Rn* \$ *j*) **using** *basis-UNIV*[*OF basis-Rm*] **by** *blast*
have *the-g*: (*THE g*. $\forall\, a$. *f* (*cart-basis-Rn* \$ *j*) \$ *a* = $(\sum x{\in}UNIV.\ g\ x *$ *cart-basis-Rm* \$ *x* \$ *a*)) = *g*
proof (*rule the-equality*, *clarify*)
fix *a*
have *f* (*cart-basis-Rn* \$ *j*) \$ *a* = $(\sum i{\in}UNIV.\ g\ i *_R$ (*cart-basis-Rm* \$ *i*)) \$ *a* **using** *setsum-g* **by** *simp*
also have ... = $(\sum x{\in}UNIV.\ g\ x *$ *cart-basis-Rm* \$ *x* \$ *a*) **unfolding** *setsum-component* **by** *simp*
finally show *f* (*cart-basis-Rn* \$ *j*) \$ *a* = $(\sum x{\in}UNIV.\ g\ x *$ *cart-basis-Rm* \$ *x* \$ *a*) **.**
fix *ga* **assume** $\forall\, a$. *f* (*cart-basis-Rn* \$ *j*) \$ *a* = $(\sum x{\in}UNIV.\ ga\ x *$ *cart-basis-Rm* \$ *x* \$ *a*)
hence *setsum-ga*: *f* (*cart-basis-Rn* \$ *j*) = $(\sum i{\in}UNIV.\ ga\ i *_R$ *cart-basis-Rm* \$ *i*) **by** (*vector*, *auto*)
show *ga* = *g*
proof (*rule basis-combination-unique*)
show *is-basis* (*set-of-vector* (*cart-basis-Rm*)) **using** *basis-Rm* **.**

show $(\sum i{\in}UNIV.\ g\ i *_R\ cart\text{-}basis\text{-}Rm\ \$\ i) = (\sum i{\in}UNIV.\ ga\ i *_R\ cart\text{-}basis\text{-}Rm\ \$\ i)$ **using** *setsum-g setsum-ga* **by** *simp*
qed
qed
show $(THE\ fa.\ \forall i.\ f\ (cart\text{-}basis\text{-}Rn\ \$\ j)\ \$\ i = (\sum x{\in}UNIV.\ fa\ x * cart\text{-}basis\text{-}Rm\ \$\ x\ \$\ i))\ i = f\ (axis\ j\ 1)\ \$\ i$
unfolding *the-g* **using** *setsum-g* **unfolding** *cart-basis-Rm cart-basis-Rn cart-basis'-def* **using** *basis-expansion-unique*[*of g f* (*axis j 1*)]
unfolding *scalar-mult-eq-scaleR* **by** *auto*
qed

lemma *matrix'*:
assumes *linear-f*: *linear f* **and** *basis-X*: *is-basis* (*set-of-vector X*) **and** *basis-Y*: *is-basis* (*set-of-vector Y*)
shows $f\ (X\$i) = setsum\ (\lambda j.\ (matrix'\ X\ Y\ f)\ \$\ j\ \$\ i *_R (Y\$j))\ UNIV$
proof (*unfold matrix'-def coord-def matrix-mult-vsum column-def*, *vector*, *auto*)
fix *j*
obtain *g* **where** g: $(\sum x{\in}UNIV.\ g\ x *_R Y\ \$\ x) = f\ (X\ \$\ i)$ **using** *basis-UNIV*[*OF basis-Y*] **by** *blast*
have *the-g*: $(THE\ fa.\ \forall ia.\ f\ (X\ \$\ i)\ \$\ ia = (\sum x{\in}UNIV.\ fa\ x * Y\ \$\ x\ \$\ ia)) = g$
proof (*rule the-equality*, *clarify*)
fix *a*
have $f\ (X\ \$\ i)\ \$\ a = (\sum x{\in}UNIV.\ g\ x *_R Y\ \$\ x)\ \$\ a$ **using** *g* **by** *simp*
also have ... $= (\sum x{\in}UNIV.\ g\ x * Y\ \$\ x\ \$\ a)$ **unfolding** *setsum-component* **by** *auto*
finally show $f\ (X\ \$\ i)\ \$\ a = (\sum x{\in}UNIV.\ g\ x * Y\ \$\ x\ \$\ a)$ **.**
fix *fa*
assume $\forall ia.\ f\ (X\ \$\ i)\ \$\ ia = (\sum x{\in}UNIV.\ fa\ x * Y\ \$\ x\ \$\ ia)$
hence $\forall ia.\ f\ (X\ \$\ i)\ \$\ ia = (\sum x{\in}UNIV.\ fa\ x *_R Y\ \$\ x)\ \$\ ia$ **unfolding** *setsum-component* **by** *simp*
hence *fa*:$f\ (X\ \$\ i) = (\sum x{\in}UNIV.\ fa\ x *_R Y\ \$\ x)$ **unfolding** *vec-eq-iff* **.**
show $fa = g$ **by** (*rule basis-combination-unique*[*OF basis-Y*], *simp add*: *fa g*)
qed
show $f\ (X\ \$\ i)\ \$\ j = (\sum x{\in}UNIV.\ (THE\ fa.\ \forall j.\ f\ (X\ \$\ i)\ \$\ j = (\sum x{\in}UNIV.\ fa\ x * Y\ \$\ x\ \$\ j))\ x * Y\ \$\ x\ \$\ j)$
unfolding *the-g* **unfolding** *g*[*symmetric*] *setsum-component* **by** *simp*
qed

corollary *matrix'2*:
assumes *linear-f*: *linear f* **and** *basis-X*: *is-basis* (*set-of-vector X*) **and** *basis-Y*: *is-basis* (*set-of-vector Y*)
and *eq-f*: $\forall i.\ f\ (X\$i) = setsum\ (\lambda j.\ A\ \$\ j\ \$\ i *_R (Y\$j))\ UNIV$
shows $matrix'\ X\ Y\ f = A$
proof −
have *eq-f'*: $\forall i.\ f\ (X\$i) = setsum\ (\lambda j.\ (matrix'\ X\ Y\ f)\ \$\ j\ \$\ i *_R (Y\$j))\ UNIV$ **using** *matrix'*[*OF linear-f basis-X basis-Y*] **by** *auto*
show *?thesis*

proof (*vector*, *auto*)
fix *i j*
def $a \equiv \lambda x.$ (*matrix′ X Y f*) \$ *x* \$ *i*
def $b \equiv \lambda x.$ *A* \$ *x* \$ *i*
have *fxi-1*:*f* (*X*\$*i*) = *setsum* ($\lambda j.$ *a j* $*_R$ (*Y*\$*j*)) *UNIV* **using** *eq-f′* **unfolding** *a-def* **by** *simp*
have *fxi-2*: *f* (*X*\$*i*) = *setsum* ($\lambda j.$ *b j* $*_R$ (*Y*\$*j*)) *UNIV* **using** *eq-f* **unfolding** *b-def* **by** *simp*
have *a*=*b* **using** *basis-combination-unique*[*OF basis-Y*] *fxi-1 fxi-2* **by** *auto*
thus (*matrix′ X Y f*) \$ *j* \$ *i* = *A* \$ *j* \$ *i* **unfolding** *a-def b-def* **by** *metis*
qed
qed

lemma *coord-matrix′*:
fixes *X*::*real*ˆ′*n*ˆ′*n* **and** *Y*::*real*ˆ′*m*ˆ′*m*
assumes *basis-X*: *is-basis* (*set-of-vector X*) **and** *basis-Y*: *is-basis* (*set-of-vector Y*) **and** *linear-f*: *linear f*
shows *coord Y* (*f v*) = (*matrix′ X Y f*) $*v$ (*coord X v*)
proof (*unfold matrix-mult-vsum matrix′-def column-def coord-def*, *vector*, *auto*)
fix *i*
obtain *g* **where** *g*: ($\sum$ *x*∈*UNIV*. *g x* $*_R$ *Y* \$ *x*) = *f v* **using** *basis-UNIV*[*OF basis-Y*] **by** *auto*
obtain *s* **where** *s*: ($\sum$ *x*∈*UNIV*. *s x* $*_R$ *X* \$ *x*) = *v* **using** *basis-UNIV*[*OF basis-X*] **by** *auto*
have *the-g*: (*THE fa*. ∀ *a*. *f v* \$ *a* = ($\sum$ *x*∈*UNIV*. *fa x* $*$ *Y* \$ *x* \$ *a*)) = *g*
proof (*rule the-equality*)
have ∀ *a*. *f v* \$ *a* = ($\sum$ *x*∈*UNIV*. *g x* $*_R$ *Y* \$ *x*) \$ *a* **using** *g* **by** *simp*
thus ∀ *a*. *f v* \$ *a* = ($\sum$ *x*∈*UNIV*. *g x* $*$ *Y* \$ *x* \$ *a*) **unfolding** *setsum-component* **by** *simp*
fix *fa* **assume** ∀ *a*. *f v* \$ *a* = ($\sum$ *x*∈*UNIV*. *fa x* $*$ *Y* \$ *x* \$ *a*)
hence *fa*: *f v* = ($\sum$ *x*∈*UNIV*. *fa x* $*_R$ *Y* \$ *x*) **by** (*vector*, *auto*)
show *fa*=*g* **by** (*rule basis-combination-unique*[*OF basis-Y*], *simp add*: *fa g*)
qed
have *the-s*: (*THE f*. ∀ *i*. *v* \$ *i* = ($\sum$ *x*∈*UNIV*. *f x* $*$ *X* \$ *x* \$ *i*))=*s*
proof (*rule the-equality*)
have ∀ *i*. *v* \$ *i* = ($\sum$ *x*∈*UNIV*. *s x* $*_R$ *X* \$ *x*) \$ *i* **using** *s* **by** *simp*
thus ∀ *i*. *v* \$ *i* = ($\sum$ *x*∈*UNIV*. *s x* $*$ *X* \$ *x* \$ *i*)**unfolding** *setsum-component* **by** *simp*
fix *fa* **assume** ∀ *i*. *v* \$ *i* = ($\sum$ *x*∈*UNIV*. *fa x* $*$ *X* \$ *x* \$ *i*)
hence *fa*: *v*=($\sum$ *x*∈*UNIV*. *fa x* $*_R$ *X* \$ *x*) **by** (*vector*, *auto*)
show *fa*=*s***by** (*rule basis-combination-unique*[*OF basis-X*], *simp add*: *fa s*)
qed
def $t \equiv \lambda x.$ ($\sum$ *i*∈*UNIV*. (*s i* $*$ (*THE fa*. *f* (*X* \$ *i*) = ($\sum$ *x*∈*UNIV*. *fa x* $*_R$ *Y* \$ *x*)) *x*))
have ($\sum$ *x*∈*UNIV*. *g x* $*_R$ *Y* \$ *x*) = *f v* **using** *g* **by** *simp*
also have ... = *f* ($\sum$ *x*∈*UNIV*. *s x* $*_R$ *X* \$ *x*) **using** *s* **by** *simp*
also have ... = ($\sum$ *x*∈*UNIV*. *s x* $*_R$ *f* (*X* \$ *x*)) **by** (*rule linear-setsum-mul*[*OF linear-f*], *simp*)

also have ... = ($\sum i{\in}UNIV.\ s\ i *_R\ setsum\ (\lambda j.\ (matrix'\ X\ Y\ f)\$j\$i *_R (Y\$j))$ $UNIV$) **using** *matrix'*[*OF linear-f basis-X basis-Y*] **by** *auto*
also have ... = ($\sum i{\in}UNIV.\ \sum x{\in}UNIV.\ s\ i *_R\ matrix'\ X\ Y\ f\ \$\ x\ \$\ i *_R\ Y$ $\$\ x$) **unfolding** *scaleR-setsum-right* **..**
also have ... = ($\sum i{\in}UNIV.\ \sum x{\in}UNIV.\ (s\ i * (THE\ fa.\ f\ (X\ \$\ i) = (\sum x{\in}UNIV.\ fa\ x *_R\ Y\ \$\ x))\ x) *_R\ Y\ \$\ x$) **unfolding** *matrix'-def* **unfolding** *coord-def* **by** *auto*
also have ... = ($\sum x{\in}UNIV.\ (\sum i{\in}UNIV.\ (s\ i * (THE\ fa.\ f\ (X\ \$\ i) = (\sum x{\in}UNIV.\ fa\ x *_R\ Y\ \$\ x))\ x) *_R\ Y\ \$\ x)$) **by** (*rule setsum-commute*)
also have ... = ($\sum x{\in}UNIV.\ (\sum i{\in}UNIV.\ (s\ i * (THE\ fa.\ f\ (X\ \$\ i) = (\sum x{\in}UNIV.\ fa\ x *_R\ Y\ \$\ x))\ x)) *_R\ Y\ \$\ x$) **unfolding** *scaleR-setsum-left* **..**
also have ... = ($\sum x{\in}UNIV.\ t\ x *_R\ Y\ \$\ x$) **unfolding** *t-def* **..**
finally have ($\sum x{\in}UNIV.\ g\ x *_R\ Y\ \$\ x) = (\sum x{\in}UNIV.\ t\ x *_R\ Y\ \$\ x$) **.**
hence $g{=}t$ **using** *basis-combination-unique*[*OF basis-Y*] **by** *simp*
thus ($THE\ fa.\ \forall i.\ f\ v\ \$\ i = (\sum x{\in}UNIV.\ fa\ x * Y\ \$\ x\ \$\ i))\ i =$
($\sum x{\in}UNIV.\ (THE\ f.\ \forall i.\ v\ \$\ i = (\sum x{\in}UNIV.\ f\ x * X\ \$\ x\ \$\ i))\ x * (THE\ fa.\ \forall i.\ f\ (X\ \$\ x)\ \$\ i = (\sum x{\in}UNIV.\ fa\ x * Y\ \$\ x\ \$\ i))\ i$)
proof (*unfold the-g the-s t-def*, *auto*)
have ($\sum x{\in}UNIV.\ s\ x * (THE\ fa.\ \forall i.\ f\ (X\ \$\ x)\ \$\ i = (\sum x{\in}UNIV.\ fa\ x * Y\ \$\ x\ \$\ i))\ i) =$
($\sum x{\in}UNIV.\ s\ x * (THE\ fa.\ \forall i.\ f\ (X\ \$\ x)\ \$\ i = (\sum x{\in}UNIV.\ fa\ x *_R\ Y\ \$\ x)\ \$\ i)\ i$) **unfolding** *setsum-component* **by** *simp*
also have ... = ($\sum x{\in}UNIV.\ s\ x * (THE\ fa.\ f\ (X\ \$\ x) = (\sum x{\in}UNIV.\ fa\ x *_R\ Y\ \$\ x))\ i$) **by** (*rule setsum-cong2*, *simp add*: *vec-eq-iff*)
finally show ($\sum ia{\in}UNIV.\ s\ ia * (THE\ fa.\ f\ (X\ \$\ ia) = (\sum x{\in}UNIV.\ fa\ x *_R\ Y\ \$\ x))\ i) = (\sum x{\in}UNIV.\ s\ x * (THE\ fa.\ \forall i.\ f\ (X\ \$\ x)\ \$\ i = (\sum x{\in}UNIV.\ fa\ x * Y\ \$\ x\ \$\ i))\ i$)
by *auto*
qed
qed

lemma *matrix'-compose*:
fixes $X{::}real\hat{\ }'n\hat{\ }'n$ **and** $Y{::}real\hat{\ }'m\hat{\ }'m$ **and** $Z{::}real\hat{\ }'p\hat{\ }'p$
assumes *basis-X*: *is-basis* (*set-of-vector X*) **and** *basis-Y*: *is-basis* (*set-of-vector Y*) **and** *basis-Z*: *is-basis* (*set-of-vector Z*)
and *linear-f*: *linear f* **and** *linear-g*: *linear g*
shows $matrix'\ X\ Z\ (g \circ f) = (matrix'\ Y\ Z\ g) ** (matrix'\ X\ Y\ f)$
proof (*unfold matrix-eq*, *clarify*)
fix $a{::}(real, 'n)\ vec$
obtain v **where** v: $a = coord\ X\ v$ **using** *bij-coord*[*OF basis-X*] **unfolding** *bij-iff* **by** *metis*
have *linear-gf*: *linear* $(g \circ f)$ **using** *linear-compose*[*OF linear-f linear-g*] **.**
have $matrix'\ X\ Z\ (g \circ f) *v\ a = matrix'\ X\ Z\ (g \circ f) *v\ (coord\ X\ v)$ **unfolding** v **..**
also have ... = $coord\ Z\ ((g \circ f)\ v)$ **unfolding** *coord-matrix'*[*OF basis-X basis-Z linear-gf*, *symmetric*] **..**
also have ... = $coord\ Z\ (g\ (f\ v))$ **unfolding** *o-def* **..**
also have ... = $(matrix'\ Y\ Z\ g) *v\ (coord\ Y\ (f\ v))$ **unfolding** *coord-matrix'*[*OF basis-Y basis-Z linear-g*] **..**

also have ... = (*matrix′ Y Z g*) $*v$ ((*matrix′ X Y f*) $*v$ (*coord X v*)) **unfolding** *coord-matrix′*[*OF basis-X basis-Y linear-f*] **..**
also have ... = ((*matrix′ Y Z g*) $**$ (*matrix′ X Y f*)) $*v$ (*coord X v*) **unfolding** *matrix-vector-mul-assoc* **..**
finally show *matrix′ X Z* ($g \circ f$) $*v$ *a* = *matrix′ Y Z g* $**$ *matrix′ X Y f* $*v$ *a* **unfolding** *v* **.**
qed

lemma *exists-linear-eq-matrix′*:
fixes *A*::*real*^′*m*^′*n* **and** *X*::*real*^′*m*^′*m* **and** *Y*::*real*^′*n*^′*n*
assumes *basis-X*: *is-basis* (*set-of-vector X*) **and** *basis-Y*: *is-basis* (*set-of-vector Y*)
shows $\exists f$. *matrix′ X Y f* = *A* $\wedge$ *linear f*
proof −
def *f* == λv. *setsum* (λj. *A* \$ *j* \$ (*THE k. v* = *X* \$ *k*) $*_R$ *Y* \$ *j*) *UNIV*
obtain *g* **where** *linear-g*: *linear g* **and** *f-eq-g*: ($\forall x \in$ (*set-of-vector X*). *g x* = *f x*) **using** *linear-independent-extend* **using** *basis-X* **unfolding** *is-basis-def* **by** *blast*
show *?thesis*
proof (*rule exI*[*of - g*], *rule conjI*)
show *matrix′ X Y g* = *A*
proof (*rule matrix′2*)
show *linear g* **using** *linear-g* **.**
show *is-basis* (*set-of-vector X*) **using** *basis-X* **.**
show *is-basis* (*set-of-vector Y*) **using** *basis-Y* **.**
show $\forall i$. *g* (*X* \$ *i*) = ($\sum j \in UNIV$. *A* \$ *j* \$ *i* $*_R$ *Y* \$ *j*)
proof (*clarify*)
fix *i*
have *the-k-eq-i*: (*THE k. X* \$ *i* = *X* \$ *k*) = *i*
proof (*rule the-equality*)
show *X* \$ *i* = *X* \$ *i* **..**
fix *k* **assume** *Xi-Xk*: *X* \$ *i* = *X* \$ *k* **show** *k* = *i* **using** *Xi-Xk basis-X inj-eq inj-op-nth* **by** *metis*
qed
have *Xi-in-X*:*X*\$*i* $\in$ (*set-of-vector X*) **unfolding** *set-of-vector-def* **by** *auto*
have *g* (*X*\$*i*) = *f* (*X*\$*i*) **using** *f-eq-g Xi-in-X* **by** *simp*
also have ... = ($\sum j \in UNIV$. *A* \$ *j* \$ (*THE k. X* \$ *i* = *X* \$ *k*) $*_R$ *Y* \$ *j*) **unfolding** *f-def* **..**
also have ... = ($\sum j \in UNIV$. *A* \$ *j* \$ *i* $*_R$ *Y* \$ *j*) **unfolding** *the-k-eq-i* **..**
finally show *g* (*X* \$ *i*) = ($\sum j \in UNIV$. *A* \$ *j* \$ *i* $*_R$ *Y* \$ *j*) **.**
qed
qed
show *linear g* **using** *linear-g* **.**
qed
qed

lemma *linear-matrix′*:

assumes *basis-Y*: *is-basis* (*set-of-vector Y*)
shows *linear* (*matrix' X Y*)
proof (*unfold linear-def*, *auto*)
fix *f g*
show *matrix' X Y* (*f* + *g*) = *matrix' X Y f* + *matrix' X Y g*
proof (*unfold matrix'-def coord-def*, *vector*, *auto*)
fix *i j*
obtain *a* **where** *a*:($\sum x \in UNIV.\ a\ x *_R Y \$ x) = f\ (X \$ j)$ **using** *basis-UNIV*[*OF basis-Y*] **by** *blast*
obtain *b* **where** *b*:($\sum x \in UNIV.\ b\ x *_R Y \$ x) = g\ (X \$ j)$ **using** *basis-UNIV*[*OF basis-Y*] **by** *blast*
obtain *c* **where** *c*: ($\sum x \in UNIV.\ c\ x *_R Y \$ x) = (f + g)\ (X \$ j)$ **using** *basis-UNIV*[*OF basis-Y*] **by** *blast*
def $d \equiv \lambda i.\ a\ i + b\ i$
have ($\sum x \in UNIV.\ c\ x *_R Y \$ x) = (f + g)\ (X \$ j)$ **using** *c* **by** *simp*
also have ... = $f\ (X \$ j) + g\ (X \$ j)$ **unfolding** *plus-fun-def* **..**
also have ... = ($\sum x \in UNIV.\ a\ x *_R Y \$ x) + (\sum x \in UNIV.\ b\ x *_R Y \$ x)$ **unfolding** *a b* **..**
also have ... = ($\sum x \in UNIV.\ (a\ x *_R Y \$ x) + b\ x *_R Y \$ x)$ **unfolding** *setsum-addf* **..**
also have ... = ($\sum x \in UNIV.\ (a\ x + b\ x) *_R Y \$ x)$ **unfolding** *scaleR-add-left* **..**
also have ... = ($\sum x \in UNIV.\ (d\ x) *_R Y \$ x)$ **unfolding** *d-def* **by** *simp*
finally have ($\sum x \in UNIV.\ c\ x *_R Y \$ x) = (\sum x \in UNIV.\ d\ x *_R Y \$ x)$ **.**
hence *c-eq-d*: *c*=*d* **using** *basis-combination-unique*[*OF basis-Y*] **by** *simp*
have *the-a*: (*THE fa*. $\forall i.\ f\ (X \$ j) \$ i = (\sum x \in UNIV.\ fa\ x * Y \$ x \$ i)) = a$
proof (*rule the-equality*)
have $\forall i.\ f\ (X \$ j) \$ i = (\sum x \in UNIV.\ a\ x *_R Y \$ x) \$ i$ **using** *a* **unfolding** *vec-eq-iff* **by** *simp*
thus $\forall i.\ f\ (X \$ j) \$ i = (\sum x \in UNIV.\ a\ x * Y \$ x \$ i)$ **unfolding** *setsum-component* **by** *simp*
fix *fa* **assume** $\forall i.\ f\ (X \$ j) \$ i = (\sum x \in UNIV.\ fa\ x * Y \$ x \$ i)$
hence $f\ (X \$ j) = (\sum x \in UNIV.\ fa\ x *_R Y \$ x)$ **unfolding** *vec-eq-iff setsum-component* **by** *simp*
thus *fa* = *a* **using** *basis-combination-unique*[*OF basis-Y*] *a* **by** *simp*
qed
have *the-b*:(*THE f*. $\forall i.\ g\ (X \$ j) \$ i = (\sum x \in UNIV.\ f\ x * Y \$ x \$ i)) = b$
proof (*rule the-equality*)
have $\forall i.\ g\ (X \$ j) \$ i = (\sum x \in UNIV.\ b\ x *_R Y \$ x) \$ i$ **using** *b* **unfolding** *vec-eq-iff* **by** *simp*
thus $\forall i.\ g\ (X \$ j) \$ i = (\sum x \in UNIV.\ b\ x * Y \$ x \$ i)$ **unfolding** *setsum-component* **by** *simp*
fix *fa* **assume** $\forall i.\ g\ (X \$ j) \$ i = (\sum x \in UNIV.\ fa\ x * Y \$ x \$ i)$
hence $g\ (X \$ j) = (\sum x \in UNIV.\ fa\ x *_R Y \$ x)$ **unfolding** *vec-eq-iff setsum-component* **by** *simp*
thus *fa* = *b* **using** *basis-combination-unique*[*OF basis-Y*] *b* **by** *simp*
qed
have *the-c*: (*THE fa*. $\forall i.\ (f + g)\ (X \$ j) \$ i = (\sum x \in UNIV.\ fa\ x * Y \$ x \$ i)) = c$

proof (*rule the-equality*)
have $\forall i.\ (f + g)\ (X\ \$\ j)\ \$\ i = (\sum x\in UNIV.\ c\ x *_R\ Y\ \$\ x)\ \$\ i$ **using** c **unfolding** *vec-eq-iff* **by** *simp*
thus $\forall i.\ (f + g)\ (X\ \$\ j)\ \$\ i = (\sum x\in UNIV.\ c\ x * Y\ \$\ x\ \$\ i)$ **unfolding** *setsum-component* **by** *simp*
fix *fa* **assume** $\forall i.\ (f + g)\ (X\ \$\ j)\ \$\ i = (\sum x\in UNIV.\ fa\ x * Y\ \$\ x\ \$\ i)$
hence $(f + g)\ (X\ \$\ j) = (\sum x\in UNIV.\ fa\ x *_R\ Y\ \$\ x)$ **unfolding** *vec-eq-iff setsum-component* **by** *simp*
thus $fa = c$ **using** *basis-combination-unique*[*OF basis-Y*] c **by** *simp*
qed
show $(THE\ fa.\ \forall i.\ (f + g)\ (X\ \$\ j)\ \$\ i = (\sum x\in UNIV.\ fa\ x * Y\ \$\ x\ \$\ i))\ i =$
$(THE\ fa.\ \forall i.\ f\ (X\ \$\ j)\ \$\ i = (\sum x\in UNIV.\ fa\ x * Y\ \$\ x\ \$\ i))\ i + (THE\ f.\ \forall i.\ g\ (X\ \$\ j)\ \$\ i = (\sum x\in UNIV.\ f\ x * Y\ \$\ x\ \$\ i))\ i$
unfolding *the-a the-b the-c* **unfolding** $a\ b\ c$ **using** *c-eq-d* **unfolding** *d-def* **by** *fast*
qed
fix c
show $matrix'\ X\ Y\ (c *_R f) = c *_R matrix'\ X\ Y\ f$
proof (*unfold matrix'-def coord-def*, *vector*, *auto*)
fix $i\ j$
obtain a **where** a:$(\sum x\in UNIV.\ a\ x *_R\ Y\ \$\ x) = f\ (X\ \$\ j)$ **using** *basis-UNIV*[*OF basis-Y*] **by** *blast*
obtain b **where** b: $(\sum x\in UNIV.\ b\ x *_R\ Y\ \$\ x) = (c *_R f)\ (X\ \$\ j)$ **using** *basis-UNIV*[*OF basis-Y*] **by** *blast*
def $d\equiv\lambda i.\ c *_R a\ i$
have *the-a*: $(THE\ fa.\ \forall i.\ f\ (X\ \$\ j)\ \$\ i = (\sum x\in UNIV.\ fa\ x * Y\ \$\ x\ \$\ i)) = a$
proof (*rule the-equality*)
have $\forall i.\ f\ (X\ \$\ j)\ \$\ i = (\sum x\in UNIV.\ a\ x *_R\ Y\ \$\ x)\ \$\ i$ **using** a **unfolding** *vec-eq-iff* **by** *simp*
thus $\forall i.\ f\ (X\ \$\ j)\ \$\ i = (\sum x\in UNIV.\ a\ x * Y\ \$\ x\ \$\ i)$ **unfolding** *setsum-component* **by** *simp*
fix *fa* **assume** $\forall i.\ f\ (X\ \$\ j)\ \$\ i = (\sum x\in UNIV.\ fa\ x * Y\ \$\ x\ \$\ i)$
hence $f\ (X\ \$\ j) = (\sum x\in UNIV.\ fa\ x *_R\ Y\ \$\ x)$ **unfolding** *vec-eq-iff setsum-component* **by** *simp*
thus $fa = a$ **using** *basis-combination-unique*[*OF basis-Y*] a **by** *simp*
qed
have *the-b*: $(THE\ fa.\ \forall i.\ (c *_R f)\ (X\ \$\ j)\ \$\ i = (\sum x\in UNIV.\ fa\ x * Y\ \$\ x\ \$\ i)) = b$
proof (*rule the-equality*)
have $\forall i.\ (c *_R f)\ (X\ \$\ j)\ \$\ i = (\sum x\in UNIV.\ b\ x *_R\ Y\ \$\ x)\ \$\ i$ **using** b **unfolding** *vec-eq-iff* **by** *simp*
thus $\forall i.\ (c *_R f)\ (X\ \$\ j)\ \$\ i = (\sum x\in UNIV.\ b\ x * Y\ \$\ x\ \$\ i)$ **unfolding** *setsum-component* **by** *simp*
fix *fa* **assume** $\forall i.\ (c *_R f)\ (X\ \$\ j)\ \$\ i = (\sum x\in UNIV.\ fa\ x * Y\ \$\ x\ \$\ i)$
hence $(c *_R f)\ (X\ \$\ j) = (\sum x\in UNIV.\ fa\ x *_R\ Y\ \$\ x)$ **unfolding** *vec-eq-iff setsum-component* **by** *simp*
thus $fa = b$ **using** *basis-combination-unique*[*OF basis-Y*] b **by** *simp*
qed
have $(\sum x\in UNIV.\ b\ x *_R\ Y\ \$\ x) = (c *_R f)\ (X\ \$\ j)$ **using** b **.**

also have ... = $c *_R f$ (X \$ j) **unfolding** *scaleR-fun-def* **..**
also have ... = $c *_R$ ($\sum x{\in}UNIV.\ a\ x *_R\ Y$ \$ x) **unfolding** *a* **..**
also have ... = ($\sum x{\in}UNIV.\ c *_R (a\ x *_R\ Y$ \$ x)) **unfolding** *scaleR-setsum-right* **..**
also have ... = ($\sum x{\in}UNIV.\ (c *_R a\ x) *_R\ Y$ \$ x) **by** *auto*
also have ... = ($\sum x{\in}UNIV.\ d\ x *_R\ Y$ \$ x) **unfolding** *d-def* **..**
finally have ($\sum x{\in}UNIV.\ b\ x *_R\ Y$ \$ x) = ($\sum x{\in}UNIV.\ d\ x *_R\ Y$ \$ x) **.**
hence $b{=}d$ **using** *basis-combination-unique*[*OF basis-Y*] *b* **by** *simp*
thus (*THE fa*. $\forall i$. ($c *_R f$) (X \$ j) \$ i = ($\sum x{\in}UNIV.\ fa\ x * Y$ \$ x \$ i)) i = $c *$ (*THE fa*. $\forall i$. f (X \$ j) \$ i = ($\sum x{\in}UNIV.\ fa\ x * Y$ \$ x \$ i)) i
unfolding *the-a the-b d-def*
by *simp*
qed
qed

lemma *matrix′-surj*:
assumes *basis-X*: *is-basis* (*set-of-vector X*) **and** *basis-Y*: *is-basis* (*set-of-vector Y*)
shows *surj* (*matrix′ X Y*)
proof (*unfold surj-def*, *clarify*)
fix *A*
show $\exists f$. A = *matrix′ X Y f*
using *exists-linear-eq-matrix′*[*OF basis-X basis-Y*, *of A*] **unfolding** *matrix′-def* **by** *auto*
qed

Properties of *matrix-change-of-basis ?X ?Y* = ($\chi i\ j$. *coord ?Y* (*?X* \$ j) \$ i).

lemma *matrix-change-of-basis-works*:
fixes *X*::*real*^′*n*^′*n* **and** *Y*::*real*^′*n*^′*n*
assumes *basis-X*: *is-basis* (*set-of-vector X*)
and *basis-Y*: *is-basis* (*set-of-vector Y*)
shows (*matrix-change-of-basis X Y*) $*v$ (*coord X v*) = (*coord Y v*)
proof (*unfold matrix-mult-vsum matrix-change-of-basis-def column-def coord-def*, *vector*, *auto*)
fix *i*
obtain *f* **where** *f*: ($\sum x{\in}UNIV.\ f\ x *_R\ Y$ \$ x) = v **using** *basis-UNIV*[*OF basis-Y*] **by** *blast*
obtain *g* **where** *g*: ($\sum x{\in}UNIV.\ g\ x *_R\ X$ \$ x) = v **using** *basis-UNIV*[*OF basis-X*] **by** *blast*
def $t{\equiv}\lambda x$. (*THE f*. X \$ x= ($\sum a{\in}UNIV.\ f\ a *_R\ Y$ \$ a))
def $w{\equiv}\lambda i$. ($\sum x{\in}UNIV.\ g\ x *_R\ t\ x\ i$)
have *the-f*:(*THE f*. $\forall i$. v \$ i = ($\sum x{\in}UNIV.\ f\ x * Y$ \$ x \$ i)) = f
proof (*rule the-equality*)
show $\forall i$. v \$ i = ($\sum x{\in}UNIV.\ f\ x * Y$ \$ x \$ i) **using** *f* **by** *auto*
fix *fa* **assume** $\forall i$. v \$ i = ($\sum x{\in}UNIV.\ fa\ x * Y$ \$ x \$ i)
hence $\forall i$. v \$ i = ($\sum x{\in}UNIV.\ fa\ x *_R\ Y$ \$ x) \$ i **unfolding** *setsum-component* **by** *simp*

hence *fa*: $v = (\sum x{\in}UNIV.\ fa\ x *_R\ Y\ \$\ x)$ **unfolding** *vec-eq-iff* **.**
show $fa = f$
using *basis-combination-unique*[*OF basis-Y*] *fa f* **by** *simp*
qed
have *the-g*: $(THE\ f.\ \forall\, i.\ v\ \$\ i = (\sum x{\in}UNIV.\ f\ x * X\ \$\ x\ \$\ i)) = g$
proof (*rule the-equality*)
show $\forall\, i.\ v\ \$\ i = (\sum x{\in}UNIV.\ g\ x * X\ \$\ x\ \$\ i)$ **using** *g* **by** *auto*
fix *fa* **assume** $\forall\, i.\ v\ \$\ i = (\sum x{\in}UNIV.\ fa\ x * X\ \$\ x\ \$\ i)$
hence $\forall\, i.\ v\ \$\ i = (\sum x{\in}UNIV.\ fa\ x *_R X\ \$\ x)\ \$\ i$ **unfolding** *setsum-component* **by** *simp*
hence *fa*: $v = (\sum x{\in}UNIV.\ fa\ x *_R\ X\ \$\ x)$ **unfolding** *vec-eq-iff* **.**
show $fa = g$
using *basis-combination-unique*[*OF basis-X*] *fa g* **by** *simp*
qed
have $(\sum x{\in}UNIV.\ f\ x *_R\ Y\ \$\ x) = (\sum x{\in}UNIV.\ g\ x *_R\ X\ \$\ x)$ **unfolding** *f g* **..**
also have ... $= (\sum x{\in}UNIV.\ g\ x *_R (setsum\ \ (\lambda i.\ (t\ x\ i) *_R\ Y\ \$\ i)\ UNIV))$ **unfolding** *t-def*
proof (*rule setsum-cong2*)
fix *x*
obtain *h* **where** *h*: $(\sum a{\in}UNIV.\ h\ a *_R\ Y\ \$\ a) = X\$x$ **using** *basis-UNIV*[*OF basis-Y*] **by** *blast*
have *the-h*: $(THE\ f.\ X\ \$\ x = (\sum a{\in}UNIV.\ f\ a *_R\ Y\ \$\ a)){=}h$
proof (*rule the-equality*)
show $X\ \$\ x = (\sum a{\in}UNIV.\ h\ a *_R\ Y\ \$\ a)$ **using** *h* **by** *simp*
fix *f* **assume** *f*: $X\ \$\ x = (\sum a{\in}UNIV.\ f\ a *_R\ Y\ \$\ a)$
show $f = h$ **using** *basis-combination-unique*[*OF basis-Y*] *f h* **by** *simp*
qed
show $g\ x *_R X\ \$\ x = g\ x *_R (\sum i{\in}UNIV.\ (THE\ f.\ X\ \$\ x = (\sum a{\in}UNIV.\ f\ a *_R\ Y\ \$\ a))\ i *_R\ Y\ \$\ i)$ **unfolding** *the-h h* **..**
qed
also have ... $= (\sum x{\in}UNIV.\ (setsum\ \ (\lambda i.\ g\ x *_R (t\ x\ i) *_R\ Y\ \$\ i)\ UNIV))$ **unfolding** *scaleR-setsum-right* **..**
also have ... $= (\sum i{\in}UNIV.\ \sum x{\in}UNIV.\ g\ x *_R\ t\ x\ i *_R\ Y\ \$\ i)$ **by** (*rule setsum-commute*)
also have ... $= (\sum i{\in}UNIV.\ (\sum x{\in}UNIV.\ g\ x *_R\ t\ x\ i) *_R\ Y\ \$\ i)$ **unfolding** *scaleR-setsum-left* **by** *auto*
finally have $(\sum x{\in}UNIV.\ f\ x *_R\ Y\ \$\ x) = (\sum i{\in}UNIV.\ (\sum x{\in}UNIV.\ g\ x *_R\ t\ x\ i) *_R\ Y\ \$\ i)$ **.**
hence $f{=}w$ **using** *basis-combination-unique*[*OF basis-Y*] **unfolding** *w-def* **by** *auto*
thus $(\sum x{\in}UNIV.\ (THE\ f.\ \forall\, i.\ v\ \$\ i = (\sum x{\in}UNIV.\ f\ x * X\ \$\ x\ \$\ i))\ x * (THE\ f.\ \forall\, i.\ X\ \$\ x\ \$\ i = (\sum x{\in}UNIV.\ f\ x * Y\ \$\ x\ \$\ i))\ i) =$
$(THE\ f.\ \forall\, i.\ v\ \$\ i = (\sum x{\in}UNIV.\ f\ x * Y\ \$\ x\ \$\ i))\ i$ **unfolding** *the-f the-g* **unfolding** *w-def t-def* **unfolding** *vec-eq-iff* **by** *auto*
qed

lemma *matrix-change-of-basis-mat-1*:
fixes X::*real*^′*n*^′*n*
assumes *basis-X*: *is-basis* (*set-of-vector X*)
shows *matrix-change-of-basis X X* = *mat 1*
proof (*unfold matrix-change-of-basis-def coord-def mat-def*, *vector*, *auto*)
fix *j*::′*n*
def $f \equiv \lambda i.$ *if i=j then 1::real else 0*
have *UNIV-rw*: $UNIV = insert\ j\ (UNIV-\{j\})$ **by** *auto*
have $(\sum x\in UNIV.\ f\ x *_R X\ \$\ x) = (\sum x\in(insert\ j\ (UNIV-\{j\})).\ f\ x *_R X\ \$\ x)$ **using** *UNIV-rw* **by** *simp*
also have ... $= (\lambda x.\ f\ x *_R X\ \$\ x)\ j + (\sum x\in(UNIV-\{j\}).\ f\ x *_R X\ \$\ x)$ **by** (*rule setsum-insert*, *simp+*)
also have ... $= X\$j + (\sum x\in(UNIV-\{j\}).\ f\ x *_R X\ \$\ x)$ **unfolding** *f-def* **by** *simp*
also have ... $= X\$j + 0$ **unfolding** *add-left-cancel f-def* **by** (*rule setsum-0*′, *simp*)
finally have *f*: $(\sum x\in UNIV.\ f\ x *_R X\ \$\ x) = X\$j$ **by** *simp*
have *the-f*: $(THE\ f.\ \forall i.\ X\ \$\ j\ \$\ i = (\sum x\in UNIV.\ f\ x * X\ \$\ x\ \$\ i)) = f$
proof (*rule the-equality*)
show $\forall i.\ X\ \$\ j\ \$\ i = (\sum x\in UNIV.\ f\ x * X\ \$\ x\ \$\ i)$ **using** *f* **unfolding** *vec-eq-iff* **unfolding** *setsum-component* **by** *simp*
fix *fa* **assume** $\forall i.\ X\ \$\ j\ \$\ i = (\sum x\in UNIV.\ fa\ x * X\ \$\ x\ \$\ i)$
hence $\forall i.\ X\ \$\ j\ \$\ i = (\sum x\in UNIV.\ fa\ x *_R X\ \$\ x)\ \$\ i$ **unfolding** *setsum-component* **by** *simp*
hence *fa*: $X\ \$\ j = (\sum x\in UNIV.\ fa\ x *_R X\ \$\ x)$ **unfolding** *vec-eq-iff* **.**
show $fa = f$ **using** *basis-combination-unique*[*OF basis-X*] *fa f* **by** *simp*
qed
show $(THE\ f.\ \forall i.\ X\ \$\ j\ \$\ i = (\sum x\in UNIV.\ f\ x * X\ \$\ x\ \$\ i))\ j = 1$ **unfolding** *the-f f-def* **by** *simp*
fix *i* **assume** *i-not-j*: $i \neq j$
show $(THE\ f.\ \forall i.\ X\ \$\ j\ \$\ i = (\sum x\in UNIV.\ f\ x * X\ \$\ x\ \$\ i))\ i = 0$ **unfolding** *the-f f-def* **using** *i-not-j* **by** *simp*
qed

Relationships between *matrix*′ *?X ?Y ?f* = $(\chi i\ j.$ *coord ?Y* (*?f* (*?X* \$ *j*)) \$ *i*) and *matrix-change-of-basis ?X ?Y* = $(\chi i\ j.$ *coord ?Y* (*?X* \$ *j*) \$ *i*)

lemma *matrix*′*-matrix-change-of-basis*:
fixes *B*::*real*^′*n*^′*n* **and** *B*′::*real*^′*n*^′*n* **and** *C*::*real*^′*m*^′*m* **and** *C*′::*real*^′*m*^′*m*
assumes *basis-B*: *is-basis* (*set-of-vector B*) **and** *basis-B*′: *is-basis* (*set-of-vector B*′)
and *basis-C*: *is-basis* (*set-of-vector C*) **and** *basis-C*′: *is-basis* (*set-of-vector C*′)
and *linear-f*: *linear f*
shows *matrix*′ *B*′ *C*′ *f* = *matrix-change-of-basis C C*′ ** *matrix*′ *B C f* ** *matrix-change-of-basis B*′ *B*
proof (*unfold matrix-eq*, *clarify*)
fix *x*
obtain *v* **where** *v*: *x*=*coord B*′ *v* **using** *bij-coord*[*OF basis-B*′] **unfolding** *bij-iff* **by** *metis*
have *matrix-change-of-basis C C*′ ** *matrix*′ *B C f*** *matrix-change-of-basis B*′

$B *v$ (*coord* B' v)
  = *matrix-change-of-basis* C C' $**$ *matrix'* B C f $*v$ (*matrix-change-of-basis* B' B $*v$ (*coord* B' v)) **unfolding** *matrix-vector-mul-assoc* **..**
 also have ... = *matrix-change-of-basis* C C' $**$ *matrix'* B C f $*v$ (*coord* B v) **unfolding** *matrix-change-of-basis-works*[*OF basis-B' basis-B*] **..**
 also have ... = *matrix-change-of-basis* C C' $*v$ (*matrix'* B C f $*v$ (*coord* B v)) **unfolding** *matrix-vector-mul-assoc* **..**
 also have ... = *matrix-change-of-basis* C C' $*v$ (*coord* C (f v)) **unfolding** *coord-matrix'*[*OF basis-B basis-C linear-f*] **..**
 also have ... = *coord* C' (f v) **unfolding** *matrix-change-of-basis-works*[*OF basis-C basis-C'*] **..**
 also have ... = *matrix'* B' C' f $*v$ *coord* B' v **unfolding** *coord-matrix'*[*OF basis-B' basis-C' linear-f*] **..**
 finally show *matrix'* B' C' f $*v$ x = *matrix-change-of-basis* C C' $**$ *matrix'* B C f $**$ *matrix-change-of-basis* B' B $*v$ x **unfolding** v **..**
qed

lemma *matrix'-id-eq-matrix-change-of-basis*:
 fixes X::*real^'n^'n* **and** Y::*real^'n^'n*
 assumes *basis-X*: *is-basis* (*set-of-vector* X) **and** *basis-Y*: *is-basis* (*set-of-vector* Y)
 shows *matrix'* X Y (*id*) = *matrix-change-of-basis* X Y
 unfolding *matrix'-def matrix-change-of-basis-def* **unfolding** *id-def* **..**

Relationships among *invertible-lf* $?f$ = (*linear* $?f$ $\wedge$ ($\exists$ g. *linear* g $\wedge$ $g \circ ?f$ = *id* $\wedge$ $?f \circ g$ = *id*)), *matrix-change-of-basis* $?X$ $?Y$ = ($\chi i\ j$. *coord* $?Y$ ($?X$ \$ j) \$ i), *matrix'* $?X$ $?Y$ $?f$ = ($\chi i\ j$. *coord* $?Y$ ($?f$ ($?X$ \$ j)) \$ i) and *invertible* $?A$ = ($\exists$ A'. $?A$ $**$ A' = *mat* (1::$?'a$) $\wedge$ A' $**$ $?A$ = *mat* (1::$?'a$)).

lemma *matrix-inv-matrix-change-of-basis*:
 fixes X::*real^'n^'n* **and** Y::*real^'n^'n*
 assumes *basis-X*: *is-basis* (*set-of-vector* X) **and** *basis-Y*: *is-basis* (*set-of-vector* Y)
 shows*matrix-change-of-basis* Y X = *matrix-inv* (*matrix-change-of-basis* X Y)
proof (*rule matrix-inv-unique*[*symmetric*])
 have *linear-id*: *linear id* **by** (*metis linear-id*)
 have (*matrix-change-of-basis* Y X) $**$ (*matrix-change-of-basis* X Y) = (*matrix'* Y X *id*) $**$ (*matrix'* X Y *id*)
  unfolding *matrix'-id-eq-matrix-change-of-basis*[*OF basis-X basis-Y*]
  unfolding *matrix'-id-eq-matrix-change-of-basis*[*OF basis-Y basis-X*] **..**
 also have ... = *matrix'* X X (*id* $\circ$ *id*) **using** *matrix'-compose*[*OF basis-X basis-Y basis-X linear-id linear-id*] **..**
 also have ... = *matrix-change-of-basis* X X **using** *matrix'-id-eq-matrix-change-of-basis*[*OF basis-X basis-X*] **unfolding** *o-def id-def* **.**
 also have ... = *mat 1* **using** *matrix-change-of-basis-mat-1*[*OF basis-X*] **.**
 finally show *matrix-change-of-basis* Y X $**$ *matrix-change-of-basis* X Y = *mat* *1* **.**
 have (*matrix-change-of-basis* X Y) $**$ (*matrix-change-of-basis* Y X) = (*matrix'* X Y *id*) $**$ (*matrix'* Y X *id*)
  unfolding *matrix'-id-eq-matrix-change-of-basis*[*OF basis-X basis-Y*]

unfolding *matrix'-id-eq-matrix-change-of-basis*[*OF basis-Y basis-X*] **..**
also have ... = *matrix' Y Y* (*id* ∘ *id*) **using** *matrix'-compose*[*OF basis-Y basis-X basis-Y linear-id linear-id*] **..**
also have ... = *matrix-change-of-basis Y Y* **using** *matrix'-id-eq-matrix-change-of-basis*[*OF basis-Y basis-Y*] **unfolding** *o-def id-def* **.**
also have ... = *mat 1* **using** *matrix-change-of-basis-mat-1*[*OF basis-Y*] **.**
finally show *matrix-change-of-basis X Y* ∗∗ *matrix-change-of-basis Y X* = *mat 1* **.**
qed

corollary *invertible-matrix-change-of-basis*:
fixes *X*::*real^'n^'n* **and** *Y*::*real^'n^'n*
assumes *basis-X*: *is-basis* (*set-of-vector X*) **and** *basis-Y*: *is-basis* (*set-of-vector Y*)
shows *invertible* (*matrix-change-of-basis X Y*)
by (*metis basis-X basis-Y invertible-left-inverse linear-id matrix'-id-eq-matrix-change-of-basis matrix'-matrix-change-of-basis matrix-change-of-basis-mat-1*)

lemma *invertible-lf-imp-invertible-matrix'*:
assumes *invertible-lf f* **and** *basis-X*: *is-basis* (*set-of-vector X*) **and** *basis-Y*: *is-basis* (*set-of-vector Y*)
shows *invertible* (*matrix' X Y f*)
by (*metis* (*lifting*) *assms*(*1*) *basis-X basis-Y invertible-lf-def invertible-lf-imp-invertible-matrix invertible-matrix-change-of-basis invertible-mult is-basis-cart-basis' matrix'-eq-matrix matrix'-matrix-change-of-basis*)

lemma *invertible-matrix'-imp-invertible-lf*:
assumes *invertible* (*matrix' X Y f*) **and** *basis-X*: *is-basis* (*set-of-vector X*)
and *linear-f*: *linear f* **and** *basis-Y*: *is-basis* (*set-of-vector Y*)
shows *invertible-lf f*
by (*metis assms*(*1*) *basis-X basis-Y id-o invertible-matrix-change-of-basis invertible-matrix-iff-invertible-lf' invertible-mult is-basis-cart-basis' linear-f linear-id*

matrix'-compose matrix'-eq-matrix matrix'-id-eq-matrix-change-of-basis o-id)

lemma *invertible-matrix-is-change-of-basis*:
assumes *invertible-P*: *invertible P* **and** *basis-X*: *is-basis* (*set-of-vector X*)
shows ∃!*Y*. *matrix-change-of-basis Y X* = *P* ∧ *is-basis* (*set-of-vector Y*)
proof (*auto*)
show ∃ *Y*. *matrix-change-of-basis Y X* = *P* ∧ *is-basis* (*set-of-vector Y*)
proof −
fix *i j*
obtain *f* **where** *P*: *P* = *matrix' X X f* **and** *linear-f*: *linear f* **using** *exists-linear-eq-matrix'*[*OF basis-X basis-X*, *of P*] **by** *blast*
show *?thesis*

proof (*rule exI*[*of* - χ *j*. *f* (*X*\$*j*)], *rule conjI*)
show *matrix-change-of-basis* (χ *j*. *f* (*X* \$ *j*)) *X* = *P* **unfolding** *matrix-change-of-basis-def P matrix'-def* **by** *vector*
have *invertible-f*: *invertible-lf f* **using** *invertible-matrix'-imp-invertible-lf*[*OF* - *basis-X linear-f basis-X*] **using** *invertible-P* **unfolding** *P* **by** *simp*
have *rw*: *set-of-vector* (χ *j*. *f* (*X* \$ *j*)) = *f*'(*set-of-vector X*) **unfolding** *set-of-vector-def* **by** *auto*
show *is-basis* (*set-of-vector* (χ *j*. *f* (*X* \$ *j*))) **unfolding** *rw* **using** *basis-image-linear*[*OF invertible-f basis-X*] **.**
qed
qed
fix *Y Z*
assume *basis-Y*:*is-basis* (*set-of-vector Y*) **and** *eq*: *matrix-change-of-basis Z X* = *matrix-change-of-basis Y X* **and** *basis-Z*: *is-basis* (*set-of-vector Z*)
have *ZY-coord*: $\forall$ *i*. *coord X* (*Z*\$*i*) = *coord X* (*Y*\$*i*) **using** *eq* **unfolding** *matrix-change-of-basis-def* **unfolding** *vec-eq-iff* **by** *vector*
show *Y*=*Z* **by** (*vector*, *metis ZY-coord coord-eq*[*OF basis-X*])
qed

Equivalent Matrices

Next definition follows the one presented in Modern Algebra by Seth Warner.

definition *equivalent-matrices A B* = ($\exists$ *P Q*. *invertible P* $\wedge$ *invertible Q* $\wedge$ *B* = (*matrix-inv P*)***A****Q*)

lemma *exists-basis*: $\exists$ *X*::*real*^'*n*^'*n*. *is-basis* (*set-of-vector X*) **using** *is-basis-cart-basis'* **by** *auto*

lemma *equivalent-implies-exist-matrix'*:
assumes *equivalent*: *equivalent-matrices A B*
shows $\exists$ *X Y X' Y' f*::*real*^'*n*$\Rightarrow$*real*^'*m*.
linear f $\wedge$ *matrix' X Y f* = *A* $\wedge$ *matrix' X' Y' f* = *B* $\wedge$ *is-basis* (*set-of-vector X*) $\wedge$ *is-basis* (*set-of-vector Y*) $\wedge$ *is-basis* (*set-of-vector X'*) $\wedge$ *is-basis* (*set-of-vector Y'*)
proof –
obtain *X*::*real*^'*n*^'*n* **where** *X*: *is-basis* (*set-of-vector X*) **using** *exists-basis* **by** *blast*
obtain *Y*::*real*^'*m*^'*m* **where** *Y*: *is-basis* (*set-of-vector Y*) **using** *exists-basis* **by** *blast*
obtain *P Q* **where** *B-PAQ*: *B*=(*matrix-inv P*)***A****Q* **and** *inv-P*: *invertible P* **and** *inv-Q*: *invertible Q* **using** *equivalent* **unfolding** *equivalent-matrices-def* **by** *auto*
obtain *f* **where** *f-A*: *matrix' X Y f* = *A* **and** *linear-f*: *linear f* **using** *exists-linear-eq-matrix'*[*OF X Y*] **by** *auto*
obtain *X'*::*real*^'*n*^'*n* **where** *X'*: *is-basis* (*set-of-vector X'*) **and** *Q*:*matrix-change-of-basis X' X* = *Q* **using** *invertible-matrix-is-change-of-basis*[*OF inv-Q X*] **by** *fast*
obtain *Y'*::*real*^'*m*^'*m* **where** *Y'*: *is-basis* (*set-of-vector Y'*) **and** *P*: *matrix-change-of-basis Y' Y* = *P* **using** *invertible-matrix-is-change-of-basis*[*OF inv-P Y*] **by** *fast*
have *matrix-inv-P*: *matrix-change-of-basis Y Y'* = *matrix-inv P* **using** *matrix-inv-matrix-change-of-basis*[*OF Y' Y*] *P* **by** *simp*

have *matrix′ X′ Y′ f* = *matrix-change-of-basis Y Y′* ** *matrix′ X Y f* ** *matrix-change-of-basis X′ X* **using** *matrix′-matrix-change-of-basis*[*OF X X′ Y Y′ linear-f*] **.**
also have ... = (*matrix-inv P*) ** *A* ** *Q* **unfolding** *matrix-inv-P f-A Q* **..**
also have ... = *B* **using** *B-PAQ* **..**
finally show *?thesis* **using** *f-A X X′ Y Y′ linear-f* **by** *fast*
qed

lemma *exist-matrix′-implies-equivalent*:
assumes *A*: *matrix′ X Y f* = *A*
and *B*: *matrix′ X′ Y′ f* = *B*
and *X*: *is-basis* (*set-of-vector X*)
and *Y*: *is-basis* (*set-of-vector Y*)
and *X′*: *is-basis* (*set-of-vector X′*)
and *Y′*: *is-basis* (*set-of-vector Y′*)
and *linear-f*: *linear f*
shows *equivalent-matrices A B*
proof (*unfold equivalent-matrices-def*, *rule exI*[*of - matrix-change-of-basis Y′ Y*], *rule exI*[*of - matrix-change-of-basis X′ X*], *auto*)
have *inv*: *matrix-change-of-basis Y Y′* = *matrix-inv* (*matrix-change-of-basis Y′ Y*) **using** *matrix-inv-matrix-change-of-basis*[*OF Y′ Y*] **.**
show *invertible* (*matrix-change-of-basis Y′ Y*) **using** *invertible-matrix-change-of-basis*[*OF Y′ Y*] **.**
show *invertible* (*matrix-change-of-basis X′ X*) **using** *invertible-matrix-change-of-basis*[*OF X′ X*] **.**
have *B* = *matrix′ X′ Y′ f* **using** *B* **..**
also have ... = *matrix-change-of-basis Y Y′* ** *matrix′ X Y f* ** *matrix-change-of-basis X′ X* **using** *matrix′-matrix-change-of-basis*[*OF X X′ Y Y′ linear-f*] **.**
finally show *B* = *matrix-inv* (*matrix-change-of-basis Y′ Y*) ** *A* ** *matrix-change-of-basis X′ X* **unfolding** *inv* **unfolding** *A* **.**
qed

corollary *equivalent-iff-exist-matrix′*:
shows *equivalent-matrices A B* ⟷ (∃ *X Y X′ Y′ f*::*real^′n⇒real^′m*.
linear f ∧ *matrix′ X Y f* = *A* ∧ *matrix′ X′ Y′ f* = *B* ∧ *is-basis* (*set-of-vector X*) ∧ *is-basis* (*set-of-vector Y*) ∧ *is-basis* (*set-of-vector X′*) ∧ *is-basis* (*set-of-vector Y′*))
by (*rule*, *auto simp add*: *exist-matrix′-implies-equivalent equivalent-implies-exist-matrix′*)

Similar matrices

definition *similar-matrices* :: *′a*::{*semiring-1*} *^′n^′n* ⇒ *′a*::{*semiring-1*} *^′n^′n* ⇒ *bool*
where *similar-matrices A B* = (∃ *P*. *invertible P* ∧ *B*=(*matrix-inv P*)***A*****P*)

lemma *similar-implies-exist-matrix′*:
fixes *A B*::*real^′n^′n*
assumes *similar*: *similar-matrices A B*

shows $\exists X\ Y\ f.$ *linear f* $\wedge$ *matrix' X X f* $=$ *A* $\wedge$ *matrix' Y Y f* $=$ *B* $\wedge$ *is-basis* (*set-of-vector X*) $\wedge$ *is-basis* (*set-of-vector Y*)
proof −
obtain *P* **where** *inv-P*: *invertible P* **and** *B-PAP*: *B*=(*matrix-inv P*)***A****P* **using** *similar* **unfolding** *similar-matrices-def* **by** *blast*
obtain *X*::*real^'n^'n* **where** *X*: *is-basis* (*set-of-vector X*) **using** *exists-basis* **by** *blast*
obtain *f* **where** *linear-f*: *linear f* **and** *A*: *matrix' X X f* = *A* **using** *exists-linear-eq-matrix'*[*OF X X*] **by** *blast*
obtain *Y*::*real^'n^'n* **where** *Y*: *is-basis* (*set-of-vector Y*) **and** *P*: *P* = *matrix-change-of-basis Y X* **using** *invertible-matrix-is-change-of-basis*[*OF inv-P X*] **by** *fast*
have *P'*: *matrix-inv P* = *matrix-change-of-basis X Y* **by** (*metis* (*lifting*) *P X Y matrix-inv-matrix-change-of-basis*)
have *B* = (*matrix-inv P*)***A****P* **using** *B-PAP* **.**
also have ... = *matrix-change-of-basis X Y* ** *matrix' X X f* ** *P* **unfolding** *P' A* **..**
also have ... = *matrix-change-of-basis X Y* ** *matrix' X X f* ** *matrix-change-of-basis Y X* **unfolding** *P* **..**
also have ... = *matrix' Y Y f* **using** *matrix'-matrix-change-of-basis*[*OF X Y X Y linear-f*] **by** *simp*
finally show *?thesis* **using** *X Y A linear-f* **by** *fast*
qed

lemma *exist-matrix'-implies-similar*:
fixes *A B*::*real^'n^'n*
assumes *linear-f*: *linear f* **and** *A*: *matrix' X X f* = *A* **and** *B*: *matrix' Y Y f* = *B* **and** *X*: *is-basis* (*set-of-vector X*) **and** *Y*: *is-basis* (*set-of-vector Y*)
shows *similar-matrices A B*
proof (*unfold similar-matrices-def*, *rule exI*[*of - matrix-change-of-basis Y X*], *rule conjI*)
have *B*=*matrix' Y Y f* **using** *B* **..**
also have ... = *matrix-change-of-basis X Y* ** *matrix' X X f* ** *matrix-change-of-basis Y X* **using** *matrix'-matrix-change-of-basis*[*OF X Y X Y linear-f*] **by** *simp*
also have ... = *matrix-inv* (*matrix-change-of-basis Y X*) ** *A* ** *matrix-change-of-basis Y X* **unfolding** *A matrix-inv-matrix-change-of-basis*[*OF Y X*] **..**
finally show *B* = *matrix-inv* (*matrix-change-of-basis Y X*) ** *A* ** *matrix-change-of-basis Y X* **.**
show *invertible* (*matrix-change-of-basis Y X*) **using** *invertible-matrix-change-of-basis*[*OF Y X*] **.**
qed

corollary *similar-iff-exist-matrix'*:
fixes *A B*::*real^'n^'n*
shows *similar-matrices A B* $\longleftrightarrow$ ($\exists X\ Y\ f.$ *linear f* $\wedge$ *matrix' X X f* $=$ *A* $\wedge$ *matrix' Y Y f* $=$ *B* $\wedge$ *is-basis* (*set-of-vector X*) $\wedge$ *is-basis* (*set-of-vector Y*))
by (*rule*, *auto simp add*: *exist-matrix'-implies-similar similar-implies-exist-matrix'*)

end
theory *Gauss-Jordan*
  imports
  Mod-Type
  Elementary-Operations
begin

4.1 Reduced Row Echelon Form

This function returns True if each position lesser than k in a column contains a zero.

definition *is-zero-row-upt-k* :: *'rows => nat =>'a::{zero} ^'columns::{mod-type} ^'rows => bool*
  where *is-zero-row-upt-k i k A* = (∀ *j::'columns*. (*to-nat j*) < *k* ⟶ *A* $ *i* $ *j* = *0*)

definition *is-zero-row* :: *'rows =>'a::{zero} ^'columns::{mod-type} ^'rows => bool*
  where *is-zero-row i A* = *is-zero-row-upt-k i* (*ncols A*) *A*

lemma *is-zero-row-upt-ncols*:
  fixes *A::'a::{zero} ^'columns::{mod-type} ^'rows*
  shows *is-zero-row-upt-k i* (*ncols A*) *A* = (∀ *j::'columns*. *A* $ *i* $ *j* = *0*) **unfolding** *is-zero-row-def is-zero-row-upt-k-def ncols-def* **by** *auto*

corollary *is-zero-row-def'*:
  fixes *A::'a::{zero} ^'columns::{mod-type} ^'rows*
  shows *is-zero-row i A* = (∀ *j::'columns*. *A* $ *i* $ *j* = *0*) **using** *is-zero-row-upt-ncols*
unfolding *is-zero-row-def ncols-def* **.**

lemma *not-is-zero-row-upt-suc*:
  assumes ¬ *is-zero-row-upt-k i* (*Suc k*) *A*
  and ∀ *i*. *A* $ *i* $ (*from-nat k*) = *0*
  shows ¬ *is-zero-row-upt-k i k A*
  using *assms from-nat-to-nat-id*
  using *is-zero-row-upt-k-def less-SucE*
  by *metis*

lemma *is-zero-row-upt-k-suc*:
  assumes *is-zero-row-upt-k i k A*
  and *A* $ *i* $ (*from-nat k*) = *0*
  shows *is-zero-row-upt-k i* (*Suc k*) *A*
  using *assms* **unfolding** *is-zero-row-upt-k-def* **using** *less-SucE to-nat-from-nat*
by *metis*

lemma *is-zero-row-utp-0*:
  shows *is-zero-row-upt-k m 0 A* **unfolding** *is-zero-row-upt-k-def* **by** *fast*

lemma *is-zero-row-utp-0'*:
  shows ∀ *m*. *is-zero-row-upt-k m 0 A* **unfolding** *is-zero-row-upt-k-def* **by** *fast*

lemma *is-zero-row-upt-k-le*:
  assumes *is-zero-row-upt-k i (Suc k) A*
  shows *is-zero-row-upt-k i k A*
  using *assms* **unfolding** *is-zero-row-upt-k-def* **by** *simp*

This definitions returns True if a matrix is in reduced row echelon form up to the column k (not included), otherwise False.

definition *reduced-row-echelon-form-upt-k* :: $'a::\{zero, one\}$ ^ $'m::\{mod\text{-}type\}$ ^ $'n::\{finite, ord, plus, one\}$ => *nat* => *bool*
  where *reduced-row-echelon-form-upt-k A k* =
  (
    $(\forall i.$ *is-zero-row-upt-k i k A* $\longrightarrow \neg (\exists j.\ j>i \wedge \neg$ *is-zero-row-upt-k j k A*$)) \wedge$
    $(\forall i. \neg$ (*is-zero-row-upt-k i k A*) $\longrightarrow A \$ i \$ (LEAST\ k.\ A \$ i \$ k \neq 0) = 1) \wedge$
    (**In the following condition, i<i+1 is assumed to avoid that row i be the latest row (in that case, i+1 would be the first row)*:*)
    $(\forall i.\ i<i+1 \wedge \neg$ (*is-zero-row-upt-k i k A*) $\wedge \neg$ (*is-zero-row-upt-k* $(i+1)$ *k A*) $\longrightarrow ((LEAST\ n.\ A \$ i \$ n \neq 0) < (LEAST\ n.\ A \$ (i+1) \$ n \neq 0))) \wedge$
    $(\forall i. \neg$ (*is-zero-row-upt-k i k A*) $\longrightarrow (\forall j.\ i \neq j \longrightarrow A \$ j \$ (LEAST\ n.\ A \$ i \$ n \neq 0) = 0))$
)

lemma *rref-upt-0*: *reduced-row-echelon-form-upt-k A 0*
  unfolding *reduced-row-echelon-form-upt-k-def is-zero-row-upt-k-def* **by** *auto*

lemma *rref-upt-condition1*:
  assumes *r*: *reduced-row-echelon-form-upt-k A k*
  shows $(\forall i.$ *is-zero-row-upt-k i k A* $\longrightarrow \neg (\exists j.\ j>i \wedge \neg$ *is-zero-row-upt-k j k A*$))$
  using *r* **unfolding** *reduced-row-echelon-form-upt-k-def* **by** *simp*

lemma *rref-upt-condition2*:
  assumes *r*: *reduced-row-echelon-form-upt-k A k*
  shows $(\forall i. \neg$ (*is-zero-row-upt-k i k A*) $\longrightarrow A \$ i \$ (LEAST\ k.\ A \$ i \$ k \neq 0) = 1)$
  using *r* **unfolding** *reduced-row-echelon-form-upt-k-def* **by** *simp*

lemma *rref-upt-condition3*:
  assumes *r*: *reduced-row-echelon-form-upt-k A k*
  shows $(\forall i.\ i<i+1 \wedge \neg$ (*is-zero-row-upt-k i k A*) $\wedge \neg$ (*is-zero-row-upt-k* $(i+1)$ *k A*) $\longrightarrow ((LEAST\ n.\ A \$ i \$ n \neq 0) < (LEAST\ n.\ A \$ (i+1) \$ n \neq 0)))$
  using *r* **unfolding** *reduced-row-echelon-form-upt-k-def* **by** *simp*

lemma *rref-upt-condition4*:
  assumes *r*: *reduced-row-echelon-form-upt-k A k*
  shows $(\forall i. \neg$ (*is-zero-row-upt-k i k A*) $\longrightarrow (\forall j.\ i \neq j \longrightarrow A \$ j \$ (LEAST\ n.\ A \$ i \$ n \neq 0) = 0))$
  using *r* **unfolding** *reduced-row-echelon-form-upt-k-def* **by** *simp*

```
lemma reduced-row-echelon-form-upt-k-intro:
  assumes (∀ i. is-zero-row-upt-k i k A ⟶ ¬ (∃ j. j>i ∧ ¬ is-zero-row-upt-k j k A))
  and (∀ i. ¬ (is-zero-row-upt-k i k A) ⟶ A $ i $ (LEAST k. A $ i $ k ≠ 0) = 1)
  and (∀ i. i<i+1 ∧ ¬ (is-zero-row-upt-k i k A) ∧ ¬ (is-zero-row-upt-k (i+1) k A)
⟶ ((LEAST n. A $ i $ n ≠ 0) < (LEAST n. A $ (i+1) $ n ≠ 0)))
  and (∀ i. ¬ (is-zero-row-upt-k i k A) ⟶ (∀ j. i ≠ j ⟶ A $ j $ (LEAST n. A $ i $ n ≠ 0) = 0))
  shows reduced-row-echelon-form-upt-k A k
  unfolding reduced-row-echelon-form-upt-k-def using assms by fast

lemma rref-suc-imp-rref:
  fixes A::'a::{semiring-1}^'n::{mod-type}^'m::{mod-type}
  assumes r: reduced-row-echelon-form-upt-k A (Suc k)
  and k-le-card: Suc k < ncols A
  shows reduced-row-echelon-form-upt-k A k
proof (rule reduced-row-echelon-form-upt-k-intro)
  show ∀ i. ¬ is-zero-row-upt-k i k A ⟶ A $ i $ (LEAST k. A $ i $ k ≠ 0) = 1
    using rref-upt-condition2[OF r] less-SucI unfolding is-zero-row-upt-k-def by blast
  show ∀ i. i < i + 1 ∧ ¬ is-zero-row-upt-k i k A ∧ ¬ is-zero-row-upt-k (i + 1) k A ⟶ (LEAST n. A $ i $ n ≠ 0) < (LEAST n. A $ (i + 1) $ n ≠ 0)
    using rref-upt-condition3[OF r] less-SucI unfolding is-zero-row-upt-k-def by blast
  show ∀ i. ¬ is-zero-row-upt-k i k A ⟶ (∀ j. i ≠ j ⟶ A $ j $ (LEAST n. A $ i $ n ≠ 0) = 0)
    using rref-upt-condition4[OF r] less-SucI unfolding is-zero-row-upt-k-def by blast
  show ∀ i. is-zero-row-upt-k i k A ⟶ ¬ (∃ j>i. ¬ is-zero-row-upt-k j k A)
  proof (clarify, rule ccontr)
    fix i j
    assume zero-i: is-zero-row-upt-k i k A
      and i-less-j: i < j
      and not-zero-j: ¬ is-zero-row-upt-k j k A
    have not-zero-j-suc: ¬ is-zero-row-upt-k j (Suc k) A
      using not-zero-j unfolding is-zero-row-upt-k-def by fastforce
    hence not-zero-i-suc: ¬ is-zero-row-upt-k i (Suc k) A
      using rref-upt-condition1[OF r] i-less-j by fast
    have not-zero-i-plus-suc: ¬ is-zero-row-upt-k (i+1) (Suc k) A
    proof (cases j=i+1)
      case True thus ?thesis using not-zero-j-suc by simp
    next
      case False
      have i+1<j by (rule Suc-less[OF i-less-j False[symmetric]])
      thus ?thesis  using rref-upt-condition1[OF r] not-zero-j-suc by blast
    qed
    from this obtain n where a: A $ (i+1) $ n ≠ 0 and n-less-suc: to-nat n < Suc k
```

unfolding *is-zero-row-upt-k-def* **by** *blast*
have $(LEAST\ n.\ A\ \$\ (i+1)\ \$\ n \neq 0) \leq n$ **by** (*rule Least-le*, *simp add*: *a*)
also have ... $\leq$ *from-nat k* **by** (*metis Suc-lessD from-nat-mono′ from-nat-to-nat-id k-le-card less-Suc-eq-le n-less-suc ncols-def*)
finally have *least-le*: $(LEAST\ n.\ A\ \$\ (i + 1)\ \$\ n \neq 0) \leq$ *from-nat k* **.**
have *least-eq-k*: $(LEAST\ n.\ A\ \$\ i\ \$\ n \neq 0) =$ *from-nat k*
proof (*rule Least-equality*)
show $A\ \$\ i\ \$$ *from-nat k* $\neq 0$ **using** *not-zero-i-suc zero-i* **unfolding** *is-zero-row-upt-k-def* **by** (*metis from-nat-to-nat-id less-SucE*)
show $\bigwedge y.\ A\ \$\ i\ \$\ y \neq 0 \Longrightarrow$ *from-nat k* $\leq y$ **by** (*metis is-zero-row-upt-k-def not-leE to-nat-le zero-i*)
qed
have *i-less*: $i<i+1$
proof (*rule Suc-le′*, *rule ccontr*)
assume $\neg\ i + 1 \neq 0$ **hence** $i+1=0$ **by** *simp*
hence $i=-1$ **by** (*metis diff-0 diff-add-cancel diff-minus-eq-add minus-one*)
hence $j <= i$ **using** *Greatest-is-minus-1* **by** *blast*
thus *False* **using** *i-less-j* **by** *fastforce*
qed
have *from-nat k* $< (LEAST\ n.\ A\ \$\ (i+1)\ \$\ n \neq 0)$
using *rref-upt-condition3*[*OF r*] *i-less not-zero-i-suc not-zero-i-plus-suc least-eq-k* **by** *fastforce*
thus *False* **using** *least-le* **by** *simp*
qed
qed

lemma *reduced-row-echelon-if-all-zero*:
assumes *all-zero*: $\forall n.$ *is-zero-row-upt-k n k A*
shows *reduced-row-echelon-form-upt-k A k*
using *assms* **unfolding** *reduced-row-echelon-form-upt-k-def is-zero-row-upt-k-def* **by** *auto*

Definition of reduced row echelon form, based on *reduced-row-echelon-form-upt-k* $A\ k = ((\forall i.$ *is-zero-row-upt-k i k A* $\longrightarrow \neg (\exists j>i. \neg$ *is-zero-row-upt-k j k A*$)) \wedge (\forall i. \neg$ *is-zero-row-upt-k i k A* $\longrightarrow A\ \$\ i\ \$\ (LEAST\ k.\ A\ \$\ i\ \$\ k \neq (0::'a)) = (1::'a)) \wedge (\forall i.\ i < i + (1::'c) \wedge \neg$ *is-zero-row-upt-k i k A* $\wedge \neg$ *is-zero-row-upt-k* $(i + (1::'c))\ k\ A \longrightarrow (LEAST\ n.\ A\ \$\ i\ \$\ n \neq (0::'a)) < (LEAST\ n.\ A\ \$\ (i + (1::'c))\ \$\ n \neq (0::'a)))) \wedge (\forall i. \neg$ *is-zero-row-upt-k i k A* $\longrightarrow (\forall j.\ i \neq j \longrightarrow A\ \$\ j\ \$\ (LEAST\ n.\ A\ \$\ i\ \$\ n \neq (0::'a)) = (0::'a))))$

definition *reduced-row-echelon-form* :: *'a*::{*zero*, *one*} ^*'m*::{*mod-type*} ^*'n*::{*finite*, *ord*, *plus*, *one*} => *bool*
where *reduced-row-echelon-form A* = *reduced-row-echelon-form-upt-k A* (*ncols A*)

Equivalence between our definition of reduced row echelon form and the one presented in Steven Roman's book: Advanced Linear Algebra.

lemma *reduced-row-echelon-form-def′*:
reduced-row-echelon-form A =

(
  $(\forall i.\ \textit{is-zero-row}\ i\ A \longrightarrow \neg\ (\exists j.\ j>i \wedge \neg\ \textit{is-zero-row}\ j\ A)) \wedge$
  $(\forall i.\ \neg\ (\textit{is-zero-row}\ i\ A) \longrightarrow A\ \$\ i\ \$\ (\textit{LEAST}\ k.\ A\ \$\ i\ \$\ k \neq 0) = 1) \wedge$
  $(\forall i.\ i<i+1 \wedge \neg\ (\textit{is-zero-row}\ i\ A) \wedge \neg\ (\textit{is-zero-row}\ (i+1)\ A) \longrightarrow ((\textit{LEAST}\ k.\ A\ \$\ i\ \$\ k \neq 0) < (\textit{LEAST}\ k.\ A\ \$\ (i+1)\ \$\ k \neq 0))) \wedge$
  $(\forall i.\ \neg\ (\textit{is-zero-row}\ i\ A) \longrightarrow (\forall j.\ i \neq j \longrightarrow A\ \$\ j\ \$\ (\textit{LEAST}\ k.\ A\ \$\ i\ \$\ k \neq 0) = 0))$
) **unfolding** *reduced-row-echelon-form-def reduced-row-echelon-form-upt-k-def is-zero-row-def*
..

lemma *rref-condition1*:
  assumes *r*: *reduced-row-echelon-form A*
  shows $(\forall i.\ \textit{is-zero-row}\ i\ A \longrightarrow \neg\ (\exists j.\ j>i \wedge \neg\ \textit{is-zero-row}\ j\ A))$ **using** *r* **unfolding** *reduced-row-echelon-form-def'* **by** *simp*

lemma *rref-condition2*:
  assumes *r*: *reduced-row-echelon-form A*
  shows $(\forall i.\ \neg\ (\textit{is-zero-row}\ i\ A) \longrightarrow A\ \$\ i\ \$\ (\textit{LEAST}\ k.\ A\ \$\ i\ \$\ k \neq 0) = 1)$
using *r* **unfolding** *reduced-row-echelon-form-def'* **by** *simp*

lemma *rref-condition3*:
  assumes *r*: *reduced-row-echelon-form A*
  shows $(\forall i.\ i<i+1 \wedge \neg\ (\textit{is-zero-row}\ i\ A) \wedge \neg\ (\textit{is-zero-row}\ (i+1)\ A) \longrightarrow ((\textit{LEAST}\ n.\ A\ \$\ i\ \$\ n \neq 0) < (\textit{LEAST}\ n.\ A\ \$\ (i+1)\ \$\ n \neq 0)))$
  using *r* **unfolding** *reduced-row-echelon-form-def'* **by** *simp*

lemma *rref-condition4*:
  assumes *r*: *reduced-row-echelon-form A*
  shows $(\forall i.\ \neg\ (\textit{is-zero-row}\ i\ A) \longrightarrow (\forall j.\ i \neq j \longrightarrow A\ \$\ j\ \$\ (\textit{LEAST}\ n.\ A\ \$\ i\ \$\ n \neq 0) = 0))$
  using *r* **unfolding** *reduced-row-echelon-form-def'* **by** *simp*

4.2 The Gauss-Jordan Algorithm

Now, a computable version of the Gauss-Jordan algorithm is presented. The output will be a matrix in reduced row echelon form. We present an algorithm in which the reduction is applied by columns

Using this definition, zeros are made in the column j of a matrix A placing the pivot entry (a nonzero element) in the position (i,j). For that, a suitable row interchange is made to achieve a non-zero entry in position (i,j). Then, this pivot entry is multiplied by its inverse to make the pivot entry equals to 1. After that, are other entries of the j-th column are eliminated by subtracting suitable multiples of the i-th row from the other rows.

definition *Gauss-Jordan-in-ij* :: *'a::{semiring-1, inverse, one, uminus}^'m^'n::{finite, ord}=> 'n=>'m=>'a^'m^'n::{finite, ord}*
where *Gauss-Jordan-in-ij A i j* = (*let n* = (*LEAST n*. $A\ \$\ n\ \$\ j \neq 0 \wedge i \leq n$);
  interchange-A = (*interchange-rows A i n*);

A' = mult-row interchange-A i (1/interchange-Aij) in
vec-lambda(% s. if s=i then A' $ s else (row-add A' s i
(−(interchange-Asj))) $ s))

The following definition makes the step of Gauss-Jordan in a column. This function receives two input parameters: the column k where the step of Gauss-Jordan must be applied and a pair (which consists of the row where the pivot should be placed in the column k and the original matrix).

definition *Gauss-Jordan-column-k* :: (*nat* $\times$ (*'a*::{*zero*,*inverse*,*uminus*,*semiring-1*} ^*'m*::{*mod-type*} ^*'n*::{*mod*

=> *nat* => (*nat* $\times$ (*'a* ^*'m*::{*mod-type*} ^*'n*::{*mod-type*}))
where *Gauss-Jordan-column-k A' k* = (*let i*=*fst A'*; *A*=(*snd A'*) *in*
if ($\forall$ *m*$\geq$(*from-nat i*). *A* $ *m* $(*from-nat k*)=*0*) $\vee$ (*i* = *nrows A*) *then* (*i*,*A*)
else (*i*+*1*, (*Gauss-Jordan-in-ij A* (*from-nat i*) (*from-nat k*))))

The following definition applies the Gauss-Jordan step from the first column up to the k one (included).

definition *Gauss-Jordan-upt-k* :: *'a*::{*inverse*,*uminus*,*semiring-1*} ^*'columns*::{*mod-type*} ^*'rows*::{*mod-type*}
=> *nat*
=> *'a* ^*'columns*::{*mod-type*} ^*'rows*::{*mod-type*}
where *Gauss-Jordan-upt-k A k* = *snd* (*foldl Gauss-Jordan-column-k* (*0*,*A*) [*0*..<*Suc*
k])

Gauss-Jordan is to apply the *Gauss-Jordan-column-k* in all columns.

definition *Gauss-Jordan* :: *'a*::{*inverse*,*uminus*,*semiring-1*} ^*'columns*::{*mod-type*} ^*'rows*::{*mod-type*}

=> *'a* ^*'columns*::{*mod-type*} ^*'rows*::{*mod-type*}
where *Gauss-Jordan A* = *Gauss-Jordan-upt-k A* ((*ncols A*) − *1*)

4.3 Properties about rref and the greatest nonzero row.

lemma *greatest-plus-one-eq-0*:
fixes *A*::*'a*::{*field*} ^*'columns*::{*mod-type*} ^*'rows*::{*mod-type*} **and** *k*::*nat*
assumes *Suc* (*to-nat* (*GREATEST' n*. $\neg$ *is-zero-row-upt-k n k A*)) = *nrows A*
shows (*GREATEST' n*. $\neg$ *is-zero-row-upt-k n k A*) + *1* = *0*
proof −
have *to-nat* (*GREATEST' R*. $\neg$ *is-zero-row-upt-k R k A*) + *1* = *card* (*UNIV* ::*'rows*
set)
using *assms* **unfolding** *nrows-def* **by** *fastforce*
thus (*GREATEST' n*. $\neg$ *is-zero-row-upt-k n k A*) + (*1*::*'rows*) = (*0*::*'rows*)
using *to-nat-plus-one-less-card* **by** *fastforce*
qed

lemma *from-nat-to-nat-greatest*:
fixes *A*::*'a*::{*zero*} ^*'columns*::{*mod-type*} ^*'rows*::{*mod-type*}
shows *from-nat* (*Suc* (*to-nat* (*GREATEST' n*. $\neg$ *is-zero-row-upt-k n k A*))) =
(*GREATEST' n*. $\neg$ *is-zero-row-upt-k n k A*) + *1*
unfolding *Suc-eq-plus1*

unfolding *to-nat-1*[**where** *?'a='rows*, *symmetric*]
unfolding *add-to-nat-def* **..**

lemma *greatest-less-zero-row*:
fixes *A*::*'a*::{*one*, *zero*} ^*'n*::{*mod-type*} ^*'m*::{*finite*,*one*,*plus*,*linorder*}
assumes *r*: *reduced-row-echelon-form-upt-k A k*
and *zero-i*: *is-zero-row-upt-k i k A*
and *not-all-zero*: ¬ (∀ *a*. *is-zero-row-upt-k a k A*)
shows (*GREATEST' m*. ¬ *is-zero-row-upt-k m k A*) < *i*
proof (*rule ccontr*)
assume *not-less-i*: ¬ (*GREATEST' m*. ¬ *is-zero-row-upt-k m k A*) < *i*
have *i-less-greatest*: *i* < (*GREATEST' m*. ¬ *is-zero-row-upt-k m k A*)
by (*metis not-less-i dual-linorder.neq-iff Greatest'I not-all-zero zero-i*)
have *is-zero-row-upt-k* (*GREATEST' m*. ¬ *is-zero-row-upt-k m k A*) *k A*
using *r zero-i i-less-greatest* **unfolding** *reduced-row-echelon-form-upt-k-def* **by**
blast
thus *False* **using** *Greatest'I-ex not-all-zero* **by** *fast*
qed

lemma *rref-suc-if-zero-below-greatest*:
fixes *A*::*'a*::{*one*, *zero*} ^*'n*::{*mod-type*} ^*'m*::{*finite*,*one*,*plus*,*linorder*}
assumes *r*: *reduced-row-echelon-form-upt-k A k*
and *not-all-zero*: ¬ (∀ *a*. *is-zero-row-upt-k a k A*)
and *all-zero-below-greatest*: ∀ *a*. *a*>(*GREATEST' m*. ¬ *is-zero-row-upt-k m k A*)
⟶ *is-zero-row-upt-k a* (*Suc k*) *A*
shows *reduced-row-echelon-form-upt-k A* (*Suc k*)
proof (*rule reduced-row-echelon-form-upt-k-intro*, *auto*)
fix *i j* **assume** *zero-i-suc*: *is-zero-row-upt-k i* (*Suc k*) *A* **and** *i-le-j*: *i* < *j*
have *zero-i*: *is-zero-row-upt-k i k A* **using** *zero-i-suc* **unfolding** *is-zero-row-upt-k-def*
by *simp*
have *i*>(*GREATEST' m*. ¬ *is-zero-row-upt-k m k A*) **by** (*rule greatest-less-zero-row*[*OF*
r zero-i not-all-zero])
hence *j*>(*GREATEST' m*. ¬ *is-zero-row-upt-k m k A*) **using** *i-le-j* **by** *simp*
thus *is-zero-row-upt-k j* (*Suc k*) *A* **using** *all-zero-below-greatest* **by** *fast*
next
fix *i* **assume** *not-zero-i*: ¬ *is-zero-row-upt-k i* (*Suc k*) *A*
show *A* $ *i* $ (*LEAST k*. *A* $ *i* $ *k* ≠ *0*) = *1*
using *greatest-less-zero-row*[*OF r - not-all-zero*] *not-zero-i r all-zero-below-greatest*
unfolding *reduced-row-echelon-form-upt-k-def*
by *fast*
next
fix *i*
assume *i*: *i* < *i* + *1* **and** *not-zero-i*: ¬ *is-zero-row-upt-k i* (*Suc k*) *A* **and**
not-zero-suc-i: ¬ *is-zero-row-upt-k* (*i* + *1*) (*Suc k*) *A*
have *not-zero-i-k*: ¬ *is-zero-row-upt-k i k A*
using *all-zero-below-greatest greatest-less-zero-row*[*OF r - not-all-zero*] *not-zero-i*
by *blast*
have *not-zero-suc-i*: ¬ *is-zero-row-upt-k* (*i*+*1*) *k A*
using *all-zero-below-greatest greatest-less-zero-row*[*OF r - not-all-zero*] *not-zero-suc-i*

by *blast*
have *aux*:($\forall$ *i j*. *i* + *1* = *j* $\wedge$ *i* < *j* $\wedge$ $\neg$ *is-zero-row-upt-k i k A* $\wedge$ $\neg$ *is-zero-row-upt-k j k A* $\longrightarrow$ (*LEAST n*. *A* \$ *i* \$ *n* $\neq$ *0*) < (*LEAST n*. *A* \$ *j* \$ *n* $\neq$ *0*))
using *r* **unfolding** *reduced-row-echelon-form-upt-k-def* **by** *fast*
show (*LEAST n*. *A* \$ *i* \$ *n* $\neq$ *0*) < (*LEAST n*. *A* \$ (*i* + *1*) \$ *n* $\neq$ *0*) **using** *aux not-zero-i-k not-zero-suc-i i* **by** *simp*
next
fix *i j* **assume** $\neg$ *is-zero-row-upt-k i* (*Suc k*) *A* **and** *i* $\neq$ *j*
thus *A* \$ *j* \$ (*LEAST n*. *A* \$ *i* \$ *n* $\neq$ *0*) = *0*
using *all-zero-below-greatest greatest-less-zero-row not-all-zero r rref-upt-condition4*
by *blast*
qed

lemma *rref-suc-if-all-rows-not-zero*:
fixes *A*::*'a*::{*one*, *zero*} ^*'n*::{*mod-type*} ^*'m*::{*finite*,*one*,*plus*,*linorder*}
assumes *r*: *reduced-row-echelon-form-upt-k A k*
and *all-not-zero*: $\forall$ *n*. $\neg$ *is-zero-row-upt-k n k A*
shows *reduced-row-echelon-form-upt-k A* (*Suc k*)
proof (*rule rref-suc-if-zero-below-greatest*)
show *reduced-row-echelon-form-upt-k A k* **using** *r* **.**
show $\neg$ ($\forall$ *a*. *is-zero-row-upt-k a k A*) **using** *all-not-zero* **by** *auto*
show $\forall$ *a*>*GREATEST'* *m*. $\neg$ *is-zero-row-upt-k m k A*. *is-zero-row-upt-k a* (*Suc k*) *A*
using *all-not-zero not-greater-Greatest'* **by** *blast*
qed

lemma *greatest-ge-nonzero-row*:
fixes *A*::*'a*::{*zero*} ^*'n*::{*mod-type*} ^*'m*::{*finite*,*linorder*}
assumes $\neg$ *is-zero-row-upt-k i k A*
shows *i* $\leq$ (*GREATEST'* *m*. $\neg$ *is-zero-row-upt-k m k A*) **using** *Greatest'-ge*[*of* (λ*m*. $\neg$ *is-zero-row-upt-k m k A*), *OF assms*] **.**

lemma *greatest-ge-nonzero-row'*:
fixes *A*::*'a*::{*zero*, *one*} ^*'n*::{*mod-type*} ^*'m*::{*finite*, *linorder*, *one*, *plus*}
assumes *r*: *reduced-row-echelon-form-upt-k A k*
and *i*: *i* $\leq$ (*GREATEST'* *m*. $\neg$ *is-zero-row-upt-k m k A*)
and *not-all-zero*: $\neg$ ($\forall$ *a*. *is-zero-row-upt-k a k A*)
shows $\neg$ *is-zero-row-upt-k i k A*
using *greatest-less-zero-row*[*OF r*] *i not-all-zero* **by** *fastforce*

corollary *row-greater-greatest-is-zero*:
fixes *A*::*'a*::{*zero*} ^*'n*::{*mod-type*} ^*'m*::{*finite*,*linorder*}
assumes (*GREATEST'* *m*. $\neg$ *is-zero-row-upt-k m k A*) < *i*
shows *is-zero-row-upt-k i k A* **using** *greatest-ge-nonzero-row assms* **by** *fastforce*

4.4 Properties of *Gauss-Jordan-in-ij*

lemma *Gauss-Jordan-in-ij-1*:

fixes *A*::′*a*::{*field*} ^′*m* ^′*n*::{*finite*, *ord*, *wellorder*}
assumes *ex*: ∃ *n*. *A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*
shows (*Gauss-Jordan-in-ij A i j*) $ *i* $ *j* = *1*
proof (*unfold Gauss-Jordan-in-ij-def Let-def mult-row-def interchange-rows-def*, *vector*, *rule divide-self*)
obtain *n* **where** *Anj*: *A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n* **using** *ex* **by** *blast*
show *A* $ (*LEAST n*. *A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*) $ *j* ≠ *0* **using** *LeastI*[*of* λ*n*. *A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n n*, *OF Anj*] **by** *simp*
qed

lemma *Gauss-Jordan-in-ij-0*:
fixes *A*::′*a*::{*field*} ^′*m* ^′*n*::{*finite*, *ord*, *wellorder*}
assumes *ex*: ∃ *n*. *A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n* **and** *a*: *a* ≠ *i*
shows (*Gauss-Jordan-in-ij A i j*) $ *a* $ *j* = *0*
proof (*unfold Gauss-Jordan-in-ij-def Let-def mult-row-def interchange-rows-def row-add-def*, *auto simp add*: *a*)
obtain *n* **where** *Anj*: *A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n* **using** *ex* **by** *blast*
have *A-least*: *A* $ (*LEAST n*. *A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*) $ *j* ≠ *0* **using** *LeastI*[*of* λ*n*. *A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n n*, *OF Anj*] **by** *simp*
thus *A* $ *i* $ *j* + − (*A* $ *i* $ *j* ∗ *A* $ (*LEAST n*. *A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*) $ *j* / *A* $ (*LEAST n*. *A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*) $ *j*) = *0* **by** *fastforce*
assume *a* ≠ (*LEAST n*. *A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*)
thus *A* $ *a* $ *j* + − (*A* $ *a* $ *j* ∗ *A* $ (*LEAST n*. *A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*) $ *j* / *A* $ (*LEAST n*. *A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*) $ *j*) = *0*
using *A-least* **by** *fastforce*
qed

corollary *Gauss-Jordan-in-ij-0*′:
fixes *A*::′*a*::{*field*} ^′*m* ^′*n*::{*finite*, *ord*, *wellorder*}
assumes *ex*: ∃ *n*. *A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*
shows ∀ *a*. *a* ≠ *i* ⟶ (*Gauss-Jordan-in-ij A i j*) $ *a* $ *j* = *0* **using** *assms Gauss-Jordan-in-ij-0* **by** *blast*

lemma *Gauss-Jordan-in-ij-preserves-previous-elements*:
fixes *A*::′*a*::{*field*} ^′*columns*::{*mod-type*} ^′*rows*::{*mod-type*}
assumes *r*: *reduced-row-echelon-form-upt-k A k*
and *not-zero-a*: ¬ *is-zero-row-upt-k a k A*
and *exists-m*: ∃ *m*. *A* $ *m* $ (*from-nat k*) ≠ *0* ∧ (*GREATEST*′ *m*. ¬ *is-zero-row-upt-k m k A*) + *1* ≤ *m*
and *Greatest-plus-1*: (*GREATEST*′ *n*. ¬ *is-zero-row-upt-k n k A*) + *1* ≠ *0*
and *j-le-k*: *to-nat j* < *k*
shows *Gauss-Jordan-in-ij A* ((*GREATEST*′ *m*. ¬ *is-zero-row-upt-k m k A*) + *1*) (*from-nat k*) $ *i* $ *j* = *A* $ *i* $ *j*
proof (*unfold Gauss-Jordan-in-ij-def Let-def interchange-rows-def mult-row-def row-add-def*, *auto*)
def *last-nonzero-row* == (*GREATEST*′ *m*. ¬ *is-zero-row-upt-k m k A*)
have *last-nonzero-row* < (*last-nonzero-row* + *1*) **by** (*rule Suc-le*′[*of last-nonzero-row*], *auto simp add*: *last-nonzero-row-def Greatest-plus-1*)
hence *zero-row*: *is-zero-row-upt-k* (*last-nonzero-row* + *1*) *k A*

using *not-le greatest-ge-nonzero-row last-nonzero-row-def* **by** *fastforce*
hence *A-greatest-0*: $A \$ (last\text{-}nonzero\text{-}row + 1) \$ j = 0$ **unfolding** *is-zero-row-upt-k-def last-nonzero-row-def* **using** *j-le-k* **by** *auto*
thus $A \$ (last\text{-}nonzero\text{-}row + 1) \$ j / A \$ (last\text{-}nonzero\text{-}row + 1) \$ from\text{-}nat\ k = A \$ (last\text{-}nonzero\text{-}row + 1) \$ j$
by *simp*
have *zero*: $A \$ (LEAST\ n.\ A \$ n \$ from\text{-}nat\ k \neq 0 \wedge (GREATEST'\ m.\ \neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ m\ k\ A) + 1 \leq n) \$ j = 0$
proof −
def *least-n* $\equiv (LEAST\ n.\ A \$ n \$ from\text{-}nat\ k \neq 0 \wedge (GREATEST'\ m.\ \neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ m\ k\ A) + 1 \leq n)$
have $\exists n.\ A \$ n \$ from\text{-}nat\ k \neq 0 \wedge (GREATEST'\ m.\ \neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ m\ k\ A) + 1 \leq n$ **by** (*metis exists-m*)
from *this* **obtain** *n* **where** *n1*: $A \$ n \$ from\text{-}nat\ k \neq 0$ **and** *n2*: $(GREATEST'\ m.\ \neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ m\ k\ A) + 1 \leq n$ **by** *blast*
have $(GREATEST'\ m.\ \neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ m\ k\ A) + 1 \leq least\text{-}n$
by (*metis* (*lifting*, *full-types*) *LeastI-ex least-n-def n1 n2*)
hence *is-zero-row-upt-k least-n k A* **using** *last-nonzero-row-def less-le rref-upt-condition1* [*OF r*] *zero-row* **by** *metis*
thus $A \$ least\text{-}n \$ j = 0$ **unfolding** *is-zero-row-upt-k-def* **using** *j-le-k* **by** *simp*
qed
show $A \$ (last\text{-}nonzero\text{-}row + 1) \$ j + - (A \$ (last\text{-}nonzero\text{-}row + 1) \$ from\text{-}nat\ k *$
$A \$ (LEAST\ n.\ A \$ n \$ from\text{-}nat\ k \neq 0 \wedge (last\text{-}nonzero\text{-}row + 1) \leq n) \$ j /$
$A \$ (LEAST\ n.\ A \$ n \$ from\text{-}nat\ k \neq 0 \wedge (last\text{-}nonzero\text{-}row + 1) \leq n) \$ from\text{-}nat\ k) =$
$A \$ (LEAST\ n.\ A \$ n \$ from\text{-}nat\ k \neq 0 \wedge (last\text{-}nonzero\text{-}row + 1) \leq n) \$ j$
unfolding *last-nonzero-row-def* [*symmetric*] **unfolding** *A-greatest-0* **unfolding** *last-nonzero-row-def* **unfolding** *zero* **by** *fastforce*
show $A \$ (LEAST\ n.\ A \$ n \$ from\text{-}nat\ k \neq 0 \wedge (GREATEST'\ m.\ \neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ m\ k\ A) + 1 \leq n) \$ j /$
$A \$ (LEAST\ n.\ A \$ n \$ from\text{-}nat\ k \neq 0 \wedge (GREATEST'\ m.\ \neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ m\ k\ A) + 1 \leq n) \$ from\text{-}nat\ k =$
$A \$ ((GREATEST'\ m.\ \neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ m\ k\ A) + 1) \$ j$ **unfolding** *zero* **using** *A-greatest-0* **unfolding** *last-nonzero-row-def* **by** *simp*
show $A \$ i \$ from\text{-}nat\ k * A \$ (LEAST\ n.\ A \$ n \$ from\text{-}nat\ k \neq 0 \wedge (GREATEST'\ m.\ \neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ m\ k\ A) + 1 \leq n) \$ j /$
$A \$ (LEAST\ n.\ A \$ n \$ from\text{-}nat\ k \neq 0 \wedge (GREATEST'\ m.\ \neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ m\ k\ A) + 1 \leq n) \$ from\text{-}nat\ k =$
0 **unfolding** *zero* **by** *auto*
qed

lemma *Gauss-Jordan-in-ij-preserves-previous-elements'*:
fixes $A::'a::\{field\}\ \hat{}\ 'columns::\{mod\text{-}type\}\ \hat{}\ 'rows::\{mod\text{-}type\}$
assumes *all-zero*: $\forall n.\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ n\ k\ A$
and *j-le-k*: $to\text{-}nat\ j < k$
and *A-nk-not-zero*: $A \$ n \$ (from\text{-}nat\ k) \neq 0$

shows *Gauss-Jordan-in-ij A 0 (from-nat k) $ i $ j = A $ i $ j*
proof (*unfold Gauss-Jordan-in-ij-def Let-def mult-row-def interchange-rows-def row-add-def*, *auto*)
have *A-0-j*: *A $ 0 $ j = 0* **using** *all-zero is-zero-row-upt-k-def j-le-k* **by** *blast*
thus *A $ 0 $ j / A $ 0 $ from-nat k = A $ 0 $ j* **by** *simp*
have *A-least-j*: *A $ (LEAST n. A $ n $ from-nat k ≠ 0 ∧ 0 ≤ n) $ j = 0* **using** *all-zero is-zero-row-upt-k-def j-le-k* **by** *blast*
show *A $ 0 $ j +
− (A $ 0 $ from-nat k ∗ A $ (LEAST n. A $ n $ from-nat k ≠ 0 ∧ 0 ≤ n) $ j /
A $ (LEAST n. A $ n $ from-nat k ≠ 0 ∧ 0 ≤ n) $ from-nat k) =
A $ (LEAST n. A $ n $ from-nat k ≠ 0 ∧ 0 ≤ n) $ j* **unfolding** *A-0-j A-least-j* **by** *fastforce*
show *A $ (LEAST n. A $ n $ from-nat k ≠ 0 ∧ 0 ≤ n) $ j / A $ (LEAST n. A $ n $ from-nat k ≠ 0 ∧ 0 ≤ n) $ from-nat k = A $ 0 $ j*
unfolding *A-least-j A-0-j* **by** *simp*
show *A $ i $ from-nat k ∗ A $ (LEAST n. A $ n $ from-nat k ≠ 0 ∧ 0 ≤ n) $ j /
A $ (LEAST n. A $ n $ from-nat k ≠ 0 ∧ 0 ≤ n) $ from-nat k = 0*
unfolding *A-least-j* **by** *simp*
qed

lemma *is-zero-after-Gauss*:
fixes *A::'a::{field} ^'n::{mod-type} ^'m::{mod-type}*
assumes *zero-a*: *is-zero-row-upt-k a k A*
and *not-zero-m*: *¬ is-zero-row-upt-k m k A*
and *r*: *reduced-row-echelon-form-upt-k A k*
and *greatest-less-ma*: *(GREATEST' n. ¬ is-zero-row-upt-k n k A) + 1 ≤ ma*
and *A-ma-k-not-zero*: *A $ ma $ from-nat k ≠ 0*
shows *is-zero-row-upt-k a k (Gauss-Jordan-in-ij A ((GREATEST' m. ¬ is-zero-row-upt-k m k A) + 1) (from-nat k))*
proof (*subst is-zero-row-upt-k-def*, *clarify*)
fix *j::'n* **assume** *j-less-k*: *to-nat j < k*
have *not-zero-g*: *(GREATEST' m. ¬ is-zero-row-upt-k m k A) + 1 ≠ 0*
proof (*rule ccontr*, *simp*)
assume *(GREATEST' m. ¬ is-zero-row-upt-k m k A) + 1 = 0*
hence *(GREATEST' m. ¬ is-zero-row-upt-k m k A) = −1* **using** *a-eq-minus-1* **by** *blast*
hence *a≤(GREATEST' m. ¬ is-zero-row-upt-k m k A)* **using** *Greatest-is-minus-1* **by** *auto*
hence *¬ is-zero-row-upt-k a k A* **using** *greatest-less-zero-row[OF r] not-zero-m* **by** *fastforce*
thus *False* **using** *zero-a* **by** *contradiction*
qed
have *Gauss-Jordan-in-ij A ((GREATEST' m. ¬ is-zero-row-upt-k m k A) + 1) (from-nat k) $ a $ j = A $ a $ j*
by (*rule Gauss-Jordan-in-ij-preserves-previous-elements[OF r not-zero-m - not-zero-g j-less-k]*, *auto intro!*: *A-ma-k-not-zero greatest-less-ma*)
also have *... = 0*

using *zero-a j-less-k* **unfolding** *is-zero-row-upt-k-def* **by** *blast*
finally show *Gauss-Jordan-in-ij A ((GREATEST' m. ¬ is-zero-row-upt-k m k A) + 1) (from-nat k) $ a $ j = 0* **.**
qed

lemma *all-zero-imp-Gauss-Jordan-column-not-zero-in-row-0*:
fixes *A::'a::{field} ^'columns::{mod-type} ^'rows::{mod-type}* **and** *k::nat*
defines *ia:ia≡(if ∀ m. is-zero-row-upt-k m k A then 0 else to-nat (GREATEST' n. ¬ is-zero-row-upt-k n k A) + 1)*
defines *B:B≡(snd (Gauss-Jordan-column-k (ia,A) k))*
assumes *all-zero: ∀ n. is-zero-row-upt-k n k A*
and *not-zero-i: ¬ is-zero-row-upt-k i (Suc k) B*
and *Amk-zero: A $ m $ from-nat k ≠ 0*
shows *i=0*
proof *(rule ccontr)*
assume *i-not-0: i ≠ 0*
have *ia2: ia = 0* **using** *ia all-zero* **by** *simp*
have *B-eq-Gauss: B = Gauss-Jordan-in-ij A 0 (from-nat k)*
unfolding *B Gauss-Jordan-column-k-def Let-def fst-conv snd-conv ia2*
using *all-zero Amk-zero least-mod-type* **unfolding** *from-nat-0 nrows-def* **by** *auto*
also have *...$ i $ (from-nat k) = 0* **proof** *(rule Gauss-Jordan-in-ij-0)*
show *∃ n. A $ n $ from-nat k ≠ 0 ∧ 0 ≤ n* **using** *Amk-zero least-mod-type* **by** *blast*
show *i ≠ 0* **using** *i-not-0* **.**
qed
finally have *B $ i $ from-nat k = 0* **.**
hence *is-zero-row-upt-k i (Suc k) B*
unfolding *B-eq-Gauss*
using *Gauss-Jordan-in-ij-preserves-previous-elements'[OF all-zero - Amk-zero]*
by *(metis all-zero is-zero-row-upt-k-def less-SucE to-nat-from-nat)*
thus *False* **using** *not-zero-i* **by** *contradiction*
qed

lemma *condition-1-part-1*:
fixes *A::'a::{field} ^'columns::{mod-type} ^'rows::{mod-type}* **and** *k::nat*
assumes *zero-column-k: ∀ m≥from-nat 0. A $ m $ from-nat k = 0*
and *all-zero: ∀ m. is-zero-row-upt-k m k A*
shows *is-zero-row-upt-k j (Suc k) A*
unfolding *is-zero-row-upt-k-def* **apply** *clarify*
proof −
fix *ja::'columns* **assume** *ja-less-suc-k: to-nat ja < Suc k*
show *A $ j $ ja = 0*
proof *(cases to-nat ja < k)*
case *True* **thus** *?thesis* **using** *all-zero* **unfolding** *is-zero-row-upt-k-def* **by** *blast*
next

```
    case False hence ja-eq-k: k = to-nat ja  using ja-less-suc-k by simp
   show ?thesis using zero-column-k unfolding ja-eq-k from-nat-to-nat-id from-nat-0
using least-mod-type by blast
  qed
qed

lemma condition-1-part-2:
  fixes A::'a::{field}^'columns::{mod-type}^'rows::{mod-type} and k::nat
  assumes j-not-zero: j ≠ 0
  and all-zero: ∀ m. is-zero-row-upt-k m k A
  and Amk-not-zero: A $ m $ from-nat k ≠ 0
  shows is-zero-row-upt-k j (Suc k) (Gauss-Jordan-in-ij A (from-nat 0) (from-nat
k))
  unfolding is-zero-row-upt-k-def apply clarify
proof −
  fix ja::'columns
  assume ja-less-suc-k: to-nat ja < Suc k
  show Gauss-Jordan-in-ij A (from-nat 0) (from-nat k) $ j $ ja = 0
  proof (cases to-nat ja < k)
    case True
    have Gauss-Jordan-in-ij A (from-nat 0) (from-nat k) $ j $ ja = A $ j $ ja
     unfolding from-nat-0 using Gauss-Jordan-in-ij-preserves-previous-elements'[OF
all-zero True Amk-not-zero] .
   also have ... = 0 using all-zero True unfolding is-zero-row-upt-k-def by blast
    finally show ?thesis .
  next
    case False hence k-eq-ja: k = to-nat ja
      using ja-less-suc-k by simp
    show Gauss-Jordan-in-ij A (from-nat 0) (from-nat k) $ j $ ja = 0
      unfolding k-eq-ja from-nat-to-nat-id
    proof (rule Gauss-Jordan-in-ij-0)
      show ∃ n. A $ n $ ja ≠ 0 ∧ from-nat 0 ≤ n
        using least-mod-type Amk-not-zero
        unfolding k-eq-ja from-nat-to-nat-id from-nat-0 by blast
      show j ≠ from-nat 0 using j-not-zero unfolding from-nat-0 .
    qed
  qed
qed

lemma condition-1-part-3:
  fixes A::'a::{field}^'columns::{mod-type}^'rows::{mod-type} and k::nat
  defines ia:ia≡(if ∀ m. is-zero-row-upt-k m k A then 0 else to-nat (GREATEST'
n. ¬ is-zero-row-upt-k n k A) + 1)
  defines B:B ≡(snd (Gauss-Jordan-column-k (ia,A) k))
  assumes rref: reduced-row-echelon-form-upt-k A k
  and i-less-j: i<j
  and not-zero-m: ¬ is-zero-row-upt-k m k A
  and zero-below-greatest: ∀ m≥(GREATEST' n. ¬ is-zero-row-upt-k n k A) + 1.
A $ m $ from-nat k = 0
```

and *zero-i-suc-k*: *is-zero-row-upt-k i* (*Suc k*) *B*
shows *is-zero-row-upt-k j* (*Suc k*) *A*
unfolding *is-zero-row-upt-k-def*
proof (*auto*)
fix *ja*::*'columns*
assume *ja-less-suc-k*: *to-nat ja* < *Suc k*
have *ia2*: *ia*=*to-nat* (*GREATEST' n*. ¬ *is-zero-row-upt-k n k A*) + *1* **unfolding** *ia* **using** *not-zero-m* **by** *presburger*
have *B-eq-A*: *B*=*A*
unfolding *B Gauss-Jordan-column-k-def Let-def fst-conv snd-conv ia2*
apply *simp*
unfolding *from-nat-to-nat-greatest* **using** *zero-below-greatest* **by** *blast*
have *zero-ikA*: *is-zero-row-upt-k i k A* **using** *zero-i-suc-k* **unfolding** *B-eq-A is-zero-row-upt-k-def* **by** *fastforce*
hence *zero-jkA*: *is-zero-row-upt-k j k A* **using** *rref-upt-condition1*[*OF rref*] *i-less-j* **by** *blast*
show *A* $ *j* $ *ja* = *0*
proof (*cases to-nat ja* < *k*)
case *True*
thus *?thesis* **using** *zero-jkA* **unfolding** *is-zero-row-upt-k-def* **by** *blast*
next
case *False*
hence *k-eq-ja*:*k* = *to-nat ja* **using** *ja-less-suc-k* **by** *auto*
have (*GREATEST' n*. ¬ *is-zero-row-upt-k n k A*) + *1* ≤ *j*
proof (*rule le-Suc*, *rule Greatest'I2*)
show ¬ *is-zero-row-upt-k m k A* **using** *not-zero-m* **.**
fix *x* **assume** *not-zero-xkA*: ¬ *is-zero-row-upt-k x k A* **show** *x* < *j*
using *rref-upt-condition1*[*OF rref*] *not-zero-xkA zero-jkA neq-iff* **by** *blast*
qed
thus *?thesis* **using** *zero-below-greatest* **unfolding** *k-eq-ja from-nat-to-nat-id is-zero-row-upt-k-def* **by** *blast*
qed
qed

lemma *condition-1-part-4*:
fixes *A*::*'a*::{*field*} ^*'columns*::{*mod-type*} ^*'rows*::{*mod-type*} **and** *k*::*nat*
defines *ia*:*ia*≡(*if* ∀ *m*. *is-zero-row-upt-k m k A then 0 else to-nat* (*GREATEST' n*. ¬ *is-zero-row-upt-k n k A*) + *1*)
defines *B*:*B* ≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
assumes *rref*: *reduced-row-echelon-form-upt-k A k*
and *zero-i-suc-k*: *is-zero-row-upt-k i* (*Suc k*) *B*
and *i-less-j*: *i*<*j*
and *not-zero-m*: ¬ *is-zero-row-upt-k m k A*
and *greatest-eq-card*: *Suc* (*to-nat* (*GREATEST' n*. ¬ *is-zero-row-upt-k n k A*)) = *nrows A*
shows *is-zero-row-upt-k j* (*Suc k*) *A*
proof −
have *ia2*: *ia*=*to-nat* (*GREATEST' n*. ¬ *is-zero-row-upt-k n k A*) + *1* **unfolding** *ia* **using** *not-zero-m* **by** *presburger*

have *B-eq-A*: *B=A*
unfolding *B Gauss-Jordan-column-k-def Let-def fst-conv snd-conv ia2*
unfolding *from-nat-to-nat-greatest* **using** *greatest-eq-card nrows-def* **by** *force*
have *rref-Suc*: *reduced-row-echelon-form-upt-k A* (*Suc k*)
proof (*rule rref-suc-if-zero-below-greatest*[*OF rref*])
show ∀ *a*>*GREATEST′ m*. ¬ *is-zero-row-upt-k m k A*. *is-zero-row-upt-k a* (*Suc k*) *A*
using *greatest-eq-card not-less-eq to-nat-less-card to-nat-mono nrows-def* **by** *metis*
show ¬ (∀ *a*. *is-zero-row-upt-k a k A*) **using** *not-zero-m* **by** *fast*
qed
show *?thesis* **using** *zero-i-suc-k* **unfolding** *B-eq-A* **using** *rref-upt-condition1*[*OF rref-Suc*] *i-less-j* **by** *fast*
qed

lemma *condition-1-part-5*:
fixes *A*::′*a*::{*field*} ^′*columns*::{*mod-type*} ^′*rows*::{*mod-type*} **and** *k*::*nat*
defines *ia*:*ia*≡(*if* ∀ *m*. *is-zero-row-upt-k m k A then 0 else to-nat* (*GREATEST′ n*. ¬ *is-zero-row-upt-k n k A*) + *1*)
defines *B*:*B* ≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
assumes *rref*: *reduced-row-echelon-form-upt-k A k*
and *zero-i-suc-k*: *is-zero-row-upt-k i* (*Suc k*) *B*
and *i-less-j*: *i*<*j*
and *not-zero-m*: ¬ *is-zero-row-upt-k m k A*
and *greatest-not-card*: *Suc* (*to-nat* (*GREATEST′ n*. ¬ *is-zero-row-upt-k n k A*)) ≠ *nrows A*
and *greatest-less-ma*: (*GREATEST′ n*. ¬ *is-zero-row-upt-k n k A*) + *1* ≤ *ma*
and *A-ma-k-not-zero*: *A* $ *ma* $ *from-nat k* ≠ *0*
shows *is-zero-row-upt-k j* (*Suc k*) (*Gauss-Jordan-in-ij A* ((*GREATEST′ n*. ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*))
proof (*subst* (*1*) *is-zero-row-upt-k-def*, *clarify*)
fix *ja*::′*columns* **assume** *ja-less-suc-k*: *to-nat ja* < *Suc k*
have *ia2*: *ia*=*to-nat* (*GREATEST′ n*. ¬ *is-zero-row-upt-k n k A*) + *1* **unfolding** *ia* **using** *not-zero-m* **by** *presburger*
have *B-eq-Gauss-ij*: *B* = *Gauss-Jordan-in-ij A* ((*GREATEST′ n*. ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*)
unfolding *B Gauss-Jordan-column-k-def*
unfolding *ia2 Let-def fst-conv snd-conv*
using *greatest-not-card greatest-less-ma A-ma-k-not-zero*
by (*auto simp add*: *from-nat-to-nat-greatest nrows-def*)
have *zero-ikA*: *is-zero-row-upt-k i k A*
proof (*unfold is-zero-row-upt-k-def*, *clarify*)
fix *a*::′*columns*
assume *a-less-k*: *to-nat a* < *k*
have *A* $ *i* $ *a* = *Gauss-Jordan-in-ij A* ((*GREATEST′ n*. ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ *i* $ *a*
proof (*rule Gauss-Jordan-in-ij-preserves-previous-elements*[*symmetric*])
show *reduced-row-echelon-form-upt-k A k* **using** *rref* **.**

show ¬ *is-zero-row-upt-k m k A* **using** *not-zero-m* **.**
show ∃ *n. A* $ *n* $ *from-nat k* ≠ *0* ∧ (*GREATEST′ n.* ¬ *is-zero-row-upt-k n k A*) + *1* ≤ *n* **using** *A-ma-k-not-zero greatest-less-ma* **by** *blast*
show (*GREATEST′ n.* ¬ *is-zero-row-upt-k n k A*) + *1* ≠ *0* **using** *suc-not-zero greatest-not-card* **unfolding** *nrows-def* **by** *simp*
show *to-nat a* < *k* **using** *a-less-k* **.**
qed
also have ... = *0* **unfolding** *B-eq-Gauss-ij*[*symmetric*] **using** *zero-i-suc-k a-less-k* **unfolding** *is-zero-row-upt-k-def* **by** *simp*
finally show *A* $ *i* $ *a* = *0* **.**
qed
hence *zero-jkA*: *is-zero-row-upt-k j k A* **using** *rref-upt-condition1*[*OF rref*] *i-less-j* **by** *blast*
show *Gauss-Jordan-in-ij A* ((*GREATEST′ n.* ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ *j* $ *ja* = *0*
proof (*cases to-nat ja* < *k*)
case *True*
have *Gauss-Jordan-in-ij A* ((*GREATEST′ n.* ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ *j* $ *ja* = *A* $ *j* $ *ja*
proof (*rule Gauss-Jordan-in-ij-preserves-previous-elements*)
show *reduced-row-echelon-form-upt-k A k* **using** *rref* **.**
show ¬ *is-zero-row-upt-k m k A* **using** *not-zero-m* **.**
show ∃ *n. A* $ *n* $ *from-nat k* ≠ *0* ∧ (*GREATEST′ n.* ¬ *is-zero-row-upt-k n k A*) + *1* ≤ *n* **using** *A-ma-k-not-zero greatest-less-ma* **by** *blast*
show (*GREATEST′ n.* ¬ *is-zero-row-upt-k n k A*) + *1* ≠ *0* **using** *suc-not-zero greatest-not-card* **unfolding** *nrows-def* **by** *simp*
show *to-nat ja* < *k* **using** *True* **.**
qed
also have ... = *0* **using** *zero-jkA True* **unfolding** *is-zero-row-upt-k-def* **by** *fast*
finally show *?thesis* **.**
next
case *False* **hence** *k-eq-ja*: *k* = *to-nat ja* **using** *ja-less-suc-k* **by** *simp*
show *?thesis*
proof (*unfold k-eq-ja from-nat-to-nat-id*, *rule Gauss-Jordan-in-ij-0*)
show ∃ *n. A* $ *n* $ *ja* ≠ *0* ∧ (*GREATEST′ n.* ¬ *is-zero-row-upt-k n* (*to-nat ja*) *A*) + *1* ≤ *n*
using *A-ma-k-not-zero greatest-less-ma k-eq-ja to-nat-from-nat* **by** *auto*
show *j* ≠ (*GREATEST′ n.* ¬ *is-zero-row-upt-k n* (*to-nat ja*) *A*) + *1*
proof (*unfold k-eq-ja*[*symmetric*], *rule ccontr*)
assume ¬ *j* ≠ (*GREATEST′ n.* ¬ *is-zero-row-upt-k n k A*) + *1*
hence *j-eq*: *j* = (*GREATEST′ n.* ¬ *is-zero-row-upt-k n k A*) + *1* **by** *fast*
hence *i* < (*GREATEST′ n.* ¬ *is-zero-row-upt-k n k A*) + *1* **using** *i-less-j* **by** *force*
hence *i-le-greatest*: *i* ≤ (*GREATEST′ n.* ¬ *is-zero-row-upt-k n k A*) **using** *le-Suc dual-linorder.not-less* **by** *auto*
hence ¬ *is-zero-row-upt-k i k A* **using** *greatest-ge-nonzero-row′*[*OF rref*] *not-zero-m* **by** *fast*
thus *False* **using** *zero-ikA* **by** *contradiction*
qed

qed
qed
qed

lemma *condition-1*:
fixes *A::'a::{field}^'columns::{mod-type}^'rows::{mod-type}* **and** *k::nat*
defines *ia:ia≡(if ∀m. is-zero-row-upt-k m k A then 0 else to-nat (GREATEST' n. ¬ is-zero-row-upt-k n k A) + 1)*
defines *B:B≡(snd (Gauss-Jordan-column-k (ia,A) k))*
assumes *rref*: *reduced-row-echelon-form-upt-k A k*
and *zero-i-suc-k*: *is-zero-row-upt-k i (Suc k) B* **and** *i-less-j*: *i < j*
shows *is-zero-row-upt-k j (Suc k) B*
proof (*unfold B Gauss-Jordan-column-k-def ia Let-def fst-conv snd-conv*, *auto*, *unfold from-nat-to-nat-greatest*)
assume *zero-k*: *∀m≥from-nat 0. A $ m $ from-nat k = 0* **and** *all-zero*: *∀m. is-zero-row-upt-k m k A*
show *is-zero-row-upt-k j (Suc k) A*
using *condition-1-part-1[OF zero-k all-zero]* **.**
next
fix *m*
assume *all-zero*: *∀m. is-zero-row-upt-k m k A* **and** *Amk-not-zero*: *A $ m $ from-nat k ≠ 0*
have *j-not-0*: *j ≠ 0* **using** *i-less-j least-mod-type not-le* **by** *blast*
show *is-zero-row-upt-k j (Suc k) (Gauss-Jordan-in-ij A (from-nat 0) (from-nat k))*
using *condition-1-part-2[OF j-not-0 all-zero Amk-not-zero]* **.**
next
fix *m* **assume** *not-zero-mkA*: *¬ is-zero-row-upt-k m k A*
and *zero-below-greatest*: *∀m≥(GREATEST' n. ¬ is-zero-row-upt-k n k A) + 1. A $ m $ from-nat k = 0*
show *is-zero-row-upt-k j (Suc k) A* **using** *condition-1-part-3[OF rref i-less-j not-zero-mkA zero-below-greatest] zero-i-suc-k*
unfolding *B ia* **.**
next
fix *m* **assume** *not-zero-m*: *¬ is-zero-row-upt-k m k A*
and *greatest-eq-card*: *Suc (to-nat (GREATEST' n. ¬ is-zero-row-upt-k n k A)) = nrows A*
show *is-zero-row-upt-k j (Suc k) A*
using *condition-1-part-4[OF rref - i-less-j not-zero-m greatest-eq-card] zero-i-suc-k*
unfolding *B ia nrows-def* **.**
next
fix *m ma*
assume *not-zero-m*: *¬ is-zero-row-upt-k m k A*
and *greatest-not-card*: *Suc (to-nat (GREATEST' n. ¬ is-zero-row-upt-k n k A)) ≠ nrows A*
and *greatest-less-ma*: *(GREATEST' n. ¬ is-zero-row-upt-k n k A) + 1 ≤ ma*
and *A-ma-k-not-zero*: *A $ ma $ from-nat k ≠ 0*
show *is-zero-row-upt-k j (Suc k) (Gauss-Jordan-in-ij A ((GREATEST' n. ¬*

is-zero-row-upt-k n k A) + *1*) (*from-nat k*))
using *condition-1-part-5*[*OF rref - i-less-j not-zero-m greatest-not-card greatest-less-ma A-ma-k-not-zero*]
using *zero-i-suc-k*
unfolding *B ia* **.**
qed

lemma *condition-2-part-1*:
fixes *A*::*'a*::{*field*} ^*'columns*::{*mod-type*} ^*'rows*::{*mod-type*} **and** *k*::*nat*
defines *ia*:*ia*≡(*if* ∀ *m*. *is-zero-row-upt-k m k A then 0 else to-nat* (*GREATEST' n*. ¬ *is-zero-row-upt-k n k A*) + *1*)
defines *B*:*B*≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
assumes *not-zero-i-suc-k*: ¬ *is-zero-row-upt-k i* (*Suc k*) *B*
and *all-zero*: ∀ *m*. *is-zero-row-upt-k m k A*
and *all-zero-k*: ∀ *m*. *A* \$ *m* \$ *from-nat k* = *0*
shows *A* \$ *i* \$ (*LEAST k*. *A* \$ *i* \$ *k* ≠ *0*) = *1*
proof −
have *ia2*: *ia* = *0* **using** *ia all-zero* **by** *simp*
have *B-eq-A*: *B*=*A* **unfolding** *B Gauss-Jordan-column-k-def Let-def fst-conv snd-conv ia2* **using** *all-zero-k* **by** *fastforce*
show *?thesis* **using** *all-zero-k condition-1-part-1*[*OF - all-zero*] *not-zero-i-suc-k* **unfolding** *B-eq-A* **by** *presburger*
qed

lemma *condition-2-part-2*:
fixes *A*::*'a*::{*field*} ^*'columns*::{*mod-type*} ^*'rows*::{*mod-type*} **and** *k*::*nat*
defines *ia*:*ia*≡(*if* ∀ *m*. *is-zero-row-upt-k m k A then 0 else to-nat* (*GREATEST' n*. ¬ *is-zero-row-upt-k n k A*) + *1*)
defines *B*:*B*≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
assumes *not-zero-i-suc-k*: ¬ *is-zero-row-upt-k i* (*Suc k*) *B*
and *all-zero*: ∀ *m*. *is-zero-row-upt-k m k A*
and *Amk-not-zero*: *A* \$ *m* \$ *from-nat k* ≠ *0*
shows *Gauss-Jordan-in-ij A 0* (*from-nat k*) \$ *i* \$ (*LEAST ka*. *Gauss-Jordan-in-ij A 0* (*from-nat k*) \$ *i* \$ *ka* ≠ *0*) = *1*
proof −
have *ia2*: *ia* = *0* **unfolding** *ia* **using** *all-zero* **by** *simp*
have *B-eq*: *B* = *Gauss-Jordan-in-ij A 0* (*from-nat k*) **unfolding** *B Gauss-Jordan-column-k-def* **unfolding** *ia2 Let-def fst-conv snd-conv*
using *Amk-not-zero least-mod-type* **unfolding** *from-nat-0 nrows-def* **by** *auto*
have *i-eq-0*: *i*=*0* **using** *Amk-not-zero B-eq all-zero condition-1-part-2 from-nat-0 not-zero-i-suc-k* **by** *metis*
have *Least-eq*: (*LEAST ka*. *Gauss-Jordan-in-ij A 0* (*from-nat k*) \$ *i* \$ *ka* ≠ *0*) = *from-nat k*
proof (*rule Least-equality*)
have *Gauss-Jordan-in-ij A 0* (*from-nat k*) \$ *0* \$ *from-nat k* = *1* **using** *Gauss-Jordan-in-ij-1 Amk-not-zero least-mod-type* **by** *blast*

thus *Gauss-Jordan-in-ij A 0 (from-nat k) \$ i \$ from-nat k* $\neq$ *0* **unfolding** *i-eq-0* **by** *simp*
fix *y* **assume** *not-zero-gauss*: *Gauss-Jordan-in-ij A 0 (from-nat k) \$ i \$ y* $\neq$ *0*
show *from-nat k* $\leq$ *y*
proof (*rule ccontr*)
assume ¬ *from-nat k* $\leq$ *y* **hence** *y*: *y* < *from-nat k* **by** *force*
have *Gauss-Jordan-in-ij A 0 (from-nat k) \$ 0 \$ y* = *A \$ 0 \$ y*
by (*rule Gauss-Jordan-in-ij-preserves-previous-elements'*[*OF all-zero to-nat-le*[*OF y*] *Amk-not-zero*])
also have ... = *0* **using** *all-zero to-nat-le*[*OF y*] **unfolding** *is-zero-row-upt-k-def* **by** *blast*
finally show *False* **using** *not-zero-gauss* **unfolding** *i-eq-0* **by** *contradiction*
qed
qed
show *?thesis* **unfolding** *Least-eq* **unfolding** *i-eq-0* **by** (*rule Gauss-Jordan-in-ij-1*, *auto intro*!: *Amk-not-zero least-mod-type*)
qed

lemma *condition-2-part-3*:
fixes *A*::*'a*::{*field*} ^*'columns*::{*mod-type*} ^*'rows*::{*mod-type*} **and** *k*::*nat*
defines *ia*:*ia*≡(*if* ∀ *m. is-zero-row-upt-k m k A then 0 else to-nat* (*GREATEST' n.* ¬ *is-zero-row-upt-k n k A*) + *1*)
defines *B*:*B*≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
assumes *rref*: *reduced-row-echelon-form-upt-k A k*
and *not-zero-i-suc-k*: ¬ *is-zero-row-upt-k i* (*Suc k*) *B*
and *not-zero-m*: ¬ *is-zero-row-upt-k m k A*
and *zero-below-greatest*: ∀ *m*≥(*GREATEST' n.* ¬ *is-zero-row-upt-k n k A*) + *1*. *A \$ m \$ from-nat k* = *0*
shows *A \$ i \$* (*LEAST k. A \$ i \$ k* $\neq$ *0*) = *1*
proof −
have *ia2*: *ia*=*to-nat* (*GREATEST' n.* ¬ *is-zero-row-upt-k n k A*) + *1* **unfolding** *ia* **using** *not-zero-m* **by** *presburger*
have *B-eq-A*: *B*=*A*
unfolding *B Gauss-Jordan-column-k-def Let-def fst-conv snd-conv ia2*
apply *simp*
unfolding *from-nat-to-nat-greatest* **using** *zero-below-greatest* **by** *blast*
show *?thesis*
proof (*cases to-nat* (*GREATEST' n.* ¬ *is-zero-row-upt-k n k A*) + *1* < *CARD*(*'rows*))
case *True*
have ¬ *is-zero-row-upt-k i k A*
proof −
have *i*<(*GREATEST' n.* ¬ *is-zero-row-upt-k n k A*) + *1*
proof (*rule ccontr*)
assume ¬ *i* < (*GREATEST' n.* ¬ *is-zero-row-upt-k n k A*) + *1*
hence *i*: (*GREATEST' n.* ¬ *is-zero-row-upt-k n k A*) + *1* $\leq$ *i* **by** *simp*
hence (*GREATEST' n.* ¬ *is-zero-row-upt-k n k A*) < *i* **using** *le-Suc' True*

```
by simp
        hence zero-i: is-zero-row-upt-k i k A using not-greater-Greatest' by blast
        hence is-zero-row-upt-k i (Suc k) A
        proof (unfold is-zero-row-upt-k-def, clarify)
          fix j::'columns
          assume to-nat j < Suc k
          thus A $ i $ j = 0
            using zero-i unfolding is-zero-row-upt-k-def using zero-below-greatest
i
            by (metis from-nat-to-nat-id le-neq-implies-less not-le not-less-eq-eq)
        qed
        thus False using not-zero-i-suc-k unfolding B-eq-A by contradiction
      qed
    hence i≤(GREATEST' n. ¬ is-zero-row-upt-k n k A) using dual-linorder.not-le
le-Suc by metis
      thus ?thesis using greatest-ge-nonzero-row'[OF rref] not-zero-m by fast
    qed
    thus ?thesis using rref-upt-condition2[OF rref] by blast
  next
    case False
    have greatest-plus-one-eq-0: (GREATEST' n. ¬ is-zero-row-upt-k n k A) + 1
= 0
      using to-nat-plus-one-less-card False by blast
    have ¬ is-zero-row-upt-k i k A
    proof (rule not-is-zero-row-upt-suc)
        show ¬ is-zero-row-upt-k i (Suc k) A using not-zero-i-suc-k unfolding
B-eq-A .
      show ∀ i. A $ i $ from-nat k = 0
        using zero-below-greatest
        unfolding greatest-plus-one-eq-0 using least-mod-type by blast
    qed
    thus ?thesis using rref-upt-condition2[OF rref] by blast
  qed
qed

lemma condition-2-part-4:
  fixes A::'a::{field} ^'columns::{mod-type} ^'rows::{mod-type} and k::nat
  assumes rref: reduced-row-echelon-form-upt-k A k
  and not-zero-m: ¬ is-zero-row-upt-k m k A
  and greatest-eq-card: Suc (to-nat (GREATEST' n. ¬ is-zero-row-upt-k n k A))
= nrows A
  shows A $ i $ (LEAST k. A $ i $ k ≠ 0) = 1
proof −
  have ¬ is-zero-row-upt-k i k A
  proof (rule ccontr, simp)
    assume zero-i: is-zero-row-upt-k i k A
    hence zero-minus-1: is-zero-row-upt-k (−1) k A
      using rref-upt-condition1[OF rref]
      using Greatest-is-minus-1 neq-le-trans by metis
```

have (*GREATEST′ n.* ¬ *is-zero-row-upt-k n k A*) + *1* = *0* **using** *greatest-plus-one-eq-0*[*OF greatest-eq-card*] **.**
hence *greatest-eq-minus-1*: (*GREATEST′ n.* ¬ *is-zero-row-upt-k n k A*) = −*1* **using** *a-eq-minus-1* **by** *fast*
have ¬ *is-zero-row-upt-k* (*GREATEST′ n.* ¬ *is-zero-row-upt-k n k A*) *k A*
by (*rule greatest-ge-nonzero-row′*[*OF rref -*], *auto intro*!: *not-zero-m*)
thus *False* **using** *zero-minus-1* **unfolding** *greatest-eq-minus-1* **by** *contradiction*
qed
thus *?thesis* **using** *rref-upt-condition2*[*OF rref*] **by** *blast*
qed

lemma *condition-2-part-5*:
fixes *A*::′*a*::{*field*} ˆ′*columns*::{*mod-type*} ˆ′*rows*::{*mod-type*} **and** *k*::*nat*
defines *ia*:*ia*≡(*if* ∀ *m. is-zero-row-upt-k m k A then 0 else to-nat* (*GREATEST′ n.* ¬ *is-zero-row-upt-k n k A*) + *1*)
defines *B*:*B*≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
assumes *rref*: *reduced-row-echelon-form-upt-k A k*
and *not-zero-i-suc-k*: ¬ *is-zero-row-upt-k i* (*Suc k*) *B*
and *not-zero-m*: ¬ *is-zero-row-upt-k m k A*
and *greatest-noteq-card*: *Suc* (*to-nat* (*GREATEST′ n.* ¬ *is-zero-row-upt-k n k A*)) ≠ *nrows A*
and *greatest-less-ma*: (*GREATEST′ n.* ¬ *is-zero-row-upt-k n k A*) + *1* ≤ *ma*
and *A-ma-k-not-zero*: *A* $ *ma* $ *from-nat k* ≠ *0*
shows *Gauss-Jordan-in-ij A* ((*GREATEST′ n.* ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ *i* $
(*LEAST ka. Gauss-Jordan-in-ij A* ((*GREATEST′ n.* ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ *i* $ *ka* ≠ *0*) = *1*
proof −
have *ia2*: *ia*=*to-nat* (*GREATEST′ n.* ¬ *is-zero-row-upt-k n k A*) + *1* **unfolding** *ia* **using** *not-zero-m* **by** *presburger*
have *B-eq-Gauss*: *B*=*Gauss-Jordan-in-ij A* ((*GREATEST′ n.* ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*)
unfolding *B Gauss-Jordan-column-k-def Let-def fst-conv snd-conv ia2*
apply *simp*
unfolding *from-nat-to-nat-greatest* **using** *greatest-noteq-card A-ma-k-not-zero greatest-less-ma* **by** *blast*
have *greatest-plus-one-not-zero*: (*GREATEST′ n.* ¬ *is-zero-row-upt-k n k A*) + *1* ≠ *0*
using *suc-not-zero greatest-noteq-card* **unfolding** *nrows-def* **by** *auto*
show *?thesis*
proof (*cases is-zero-row-upt-k i k A*)
case *True*
hence *not-zero-iB*: *is-zero-row-upt-k i k B* **unfolding** *is-zero-row-upt-k-def* **unfolding** *B-eq-Gauss*
using *Gauss-Jordan-in-ij-preserves-previous-elements*[*OF rref not-zero-m - greatest-plus-one-not-zero*]
using *A-ma-k-not-zero greatest-less-ma* **by** *fastforce*
hence *Gauss-Jordan-i-not-0*: *Gauss-Jordan-in-ij A* ((*GREATEST′ n.* ¬ *is-zero-row-upt-k*

n k A) + *1*) (*from-nat k*) $ *i* $ (*from-nat k*) $\neq$ *0*
using *not-zero-i-suc-k* **unfolding** *B-eq-Gauss* **unfolding** *is-zero-row-upt-k-def*
using *from-nat-to-nat-id less-Suc-eq* **by** (*metis* (*lifting*, *no-types*))
have *i* = ((*GREATEST' n.* ¬ *is-zero-row-upt-k n k A*) + *1*)
proof (*rule ccontr*)
assume *i-not-greatest*: *i* $\neq$ (*GREATEST' n.* ¬ *is-zero-row-upt-k n k A*) + *1*
have *Gauss-Jordan-in-ij A* ((*GREATEST' n.* ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ *i* $ (*from-nat k*) = *0*
proof (*rule Gauss-Jordan-in-ij-0*)
show ∃ *n*. *A* $ *n* $ *from-nat k* $\neq$ *0* ∧ (*GREATEST' n.* ¬ *is-zero-row-upt-k n k A*) + *1* $\leq$ *n* **using** *A-ma-k-not-zero greatest-less-ma* **by** *blast*
show *i* $\neq$ (*GREATEST' n.* ¬ *is-zero-row-upt-k n k A*) + *1* **using** *i-not-greatest* **.**
qed
thus *False* **using** *Gauss-Jordan-i-not-0* **by** *contradiction*
qed
hence *Gauss-Jordan-i-1*: *Gauss-Jordan-in-ij A* ((*GREATEST' n.* ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ *i* $ (*from-nat k*) = *1*
using *Gauss-Jordan-in-ij-1* **using** *A-ma-k-not-zero greatest-less-ma* **by** *blast*
have *Least-eq-k*: (*LEAST ka. Gauss-Jordan-in-ij A* ((*GREATEST' n.* ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ *i* $ *ka* $\neq$ *0*) = *from-nat k*
proof (*rule Least-equality*)
show *Gauss-Jordan-in-ij A* ((*GREATEST' n.* ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ *i* $ *from-nat k* $\neq$ *0* **using** *Gauss-Jordan-i-not-0* **.**
show $\bigwedge$*y*. *Gauss-Jordan-in-ij A* ((*GREATEST' n.* ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ *i* $ *y* $\neq$ *0* $\Longrightarrow$ *from-nat k* $\leq$ *y*
using *B-eq-Gauss is-zero-row-upt-k-def not-less not-zero-iB to-nat-le* **by** *fast*
qed
show *?thesis* **using** *Gauss-Jordan-i-1* **unfolding** *Least-eq-k* **.**
next
case *False*
obtain *j* **where** *Aij-not-0*: *A* $ *i* $ *j* $\neq$ *0* **and** *j-le-k*: *to-nat j* < *k* **using** *False* **unfolding** *is-zero-row-upt-k-def* **by** *auto*
have *least-le-k*: *to-nat* (*LEAST ka. A* $ *i* $ *ka* $\neq$ *0*) < *k*
by (*metis* (*lifting*, *mono-tags*) *Aij-not-0 j-le-k less-trans linorder-cases not-less-Least to-nat-mono*)
have *least-le-j*: (*LEAST ka. Gauss-Jordan-in-ij A* ((*GREATEST' n.* ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ *i* $ *ka* $\neq$ *0*) $\leq$ *j*
using *Gauss-Jordan-in-ij-preserves-previous-elements*[*OF rref not-zero-m - greatest-plus-one-not-zero j-le-k*] **using** *A-ma-k-not-zero greatest-less-ma*
using *Aij-not-0 False dual-linorder.not-leE not-less-Least* **by** (*metis* (*mono-tags*))
have *Least-eq*: (*LEAST ka. Gauss-Jordan-in-ij A* ((*GREATEST' n.* ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ *i* $ *ka* $\neq$ *0*)
= (*LEAST ka. A* $ *i* $ *ka* $\neq$ *0*)
proof (*rule Least-equality*)
show *Gauss-Jordan-in-ij A* ((*GREATEST' n.* ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ *i* $ (*LEAST ka. A* $ *i* $ *ka* $\neq$ *0*) $\neq$ *0*
using *Gauss-Jordan-in-ij-preserves-previous-elements*[*OF rref False - greatest-plus-one-not-zero*] *least-le-k False rref-upt-condition2*[*OF rref*]

```
      using A-ma-k-not-zero B-eq-Gauss greatest-less-ma zero-neq-one by fastforce
        fix y assume Gauss-Jordan-y:Gauss-Jordan-in-ij A ((GREATEST' n. ¬
is-zero-row-upt-k n k A) + 1) (from-nat k) $ i $ y ≠ 0
      show (LEAST ka. A $ i $ ka ≠ 0) ≤ y
      proof (cases to-nat y < k)
        case False
        thus ?thesis
          using least-le-k less-trans not-leE to-nat-from-nat to-nat-le by metis
      next
        case True
      have A $ i $ y ≠ 0 using Gauss-Jordan-y using Gauss-Jordan-in-ij-preserves-previous-elements[OF
rref not-zero-m - greatest-plus-one-not-zero True]
          using A-ma-k-not-zero greatest-less-ma by fastforce
        thus ?thesis using Least-le by fastforce
      qed
    qed
   have A $ i $ (LEAST ka. Gauss-Jordan-in-ij A ((GREATEST' n. ¬ is-zero-row-upt-k
n k A) + 1) (from-nat k) $ i $ ka ≠ 0) = 1
      using False using rref-upt-condition2[OF rref] unfolding Least-eq by blast

   thus ?thesis unfolding Least-eq using Gauss-Jordan-in-ij-preserves-previous-elements[OF
rref False - greatest-plus-one-not-zero]
      using least-le-k A-ma-k-not-zero greatest-less-ma by fastforce
  qed
qed


lemma condition-2:
  fixes A::'a::{field} ^'columns::{mod-type} ^'rows::{mod-type} and k::nat
  defines ia:ia≡(if ∀ m. is-zero-row-upt-k m k A then 0 else to-nat (GREATEST'
n. ¬ is-zero-row-upt-k n k A) + 1)
  defines B:B≡(snd (Gauss-Jordan-column-k (ia,A) k))
  assumes rref: reduced-row-echelon-form-upt-k A k
  and not-zero-i-suc-k: ¬ is-zero-row-upt-k i (Suc k) B
  shows B $ i $ (LEAST k. B $ i $ k ≠ 0) = 1
  unfolding B Gauss-Jordan-column-k-def unfolding ia Let-def fst-conv snd-conv
apply auto unfolding from-nat-to-nat-greatest from-nat-0
proof −
  assume all-zero: ∀ m. is-zero-row-upt-k m k A and all-zero-k: ∀ m≥0. A $ m $
from-nat k = 0
  show A $ i $ (LEAST k. A $ i $ k ≠ 0) = 1
   using condition-2-part-1[OF - all-zero] not-zero-i-suc-k all-zero-k least-mod-type
unfolding B ia by blast
next
  fix m assume all-zero: ∀ m. is-zero-row-upt-k m k A
    and Amk-not-zero: A $ m $ from-nat k ≠ 0
  show Gauss-Jordan-in-ij A 0 (from-nat k) $ i $ (LEAST ka. Gauss-Jordan-in-ij
A 0 (from-nat k) $ i $ ka ≠ 0) = 1
    using condition-2-part-2[OF - all-zero Amk-not-zero] not-zero-i-suc-k unfold-
```

ing *B ia* **.**
next
  fix *m*
  assume *not-zero-m*: ¬ *is-zero-row-upt-k m k A*
    and *zero-below-greatest*: ∀ *m*≥(*GREATEST′ n.* ¬ *is-zero-row-upt-k n k A*) + *1*. *A* $ *m* $ *from-nat k* = *0*
  show *A* $ *i* $ (*LEAST k. A* $ *i* $ *k* ≠ *0*) = *1* **using** *condition-2-part-3*[*OF rref - not-zero-m zero-below-greatest*] *not-zero-i-suc-k* **unfolding** *B ia* **.**
next
  fix *m*
  assume *not-zero-m*: ¬ *is-zero-row-upt-k m k A*
    and *greatest-eq-card*: *Suc* (*to-nat* (*GREATEST′ n.* ¬ *is-zero-row-upt-k n k A*)) = *nrows A*
  show *A* $ *i* $ (*LEAST k. A* $ *i* $ *k* ≠ *0*) = *1* **using** *condition-2-part-4*[*OF rref not-zero-m greatest-eq-card*] **.**
next
  fix *m ma*
  assume *not-zero-m*: ¬ *is-zero-row-upt-k m k A*
    and *greatest-noteq-card*: *Suc* (*to-nat* (*GREATEST′ n.* ¬ *is-zero-row-upt-k n k A*)) ≠ *nrows A*
    and *greatest-less-ma*: (*GREATEST′ n.* ¬ *is-zero-row-upt-k n k A*) + *1* ≤ *ma*
    and *A-ma-k-not-zero*: *A* $ *ma* $ *from-nat k* ≠ *0*
  show *Gauss-Jordan-in-ij A* ((*GREATEST′ n.* ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ *i*
    $ (*LEAST ka. Gauss-Jordan-in-ij A* ((*GREATEST′ n.* ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ *i* $ *ka* ≠ *0*) = *1*
   using *condition-2-part-5*[*OF rref - not-zero-m greatest-noteq-card greatest-less-ma A-ma-k-not-zero*] *not-zero-i-suc-k* **unfolding** *B ia* **.**
qed

lemma *condition-3-part-1*:
  fixes *A*::′*a*::{*field*} ^′*columns*::{*mod-type*} ^′*rows*::{*mod-type*} **and** *k*::*nat*
  defines *ia*:*ia*≡(*if* ∀ *m. is-zero-row-upt-k m k A then 0 else to-nat* (*GREATEST′ n.* ¬ *is-zero-row-upt-k n k A*) + *1*)
  defines *B*:*B*≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
  assumes *not-zero-i-suc-k*: ¬ *is-zero-row-upt-k i* (*Suc k*) *B*
  and *all-zero*: ∀ *m. is-zero-row-upt-k m k A*
  and *all-zero-k*: ∀ *m. A* $ *m* $ *from-nat k* = *0*
  shows (*LEAST n. A* $ *i* $ *n* ≠ *0*) < (*LEAST n. A* $ (*i* + *1*) $ *n* ≠ *0*)
proof −
  have *ia2*: *ia* = *0* **using** *ia all-zero* **by** *simp*
  have *B-eq-A*: *B*=*A* **unfolding** *B Gauss-Jordan-column-k-def Let-def fst-conv snd-conv ia2* **using** *all-zero-k* **by** *fastforce*
  have *is-zero-row-upt-k i* (*Suc k*) *B* **using** *all-zero all-zero-k* **unfolding** *B-eq-A is-zero-row-upt-k-def* **by** (*metis less-SucE to-nat-from-nat*)
  thus *?thesis* **using** *not-zero-i-suc-k* **by** *contradiction*
qed

lemma *condition-3-part-2*:
  fixes *A*::'*a*::{*field*} ^'*columns*::{*mod-type*} ^'*rows*::{*mod-type*} **and** *k*::*nat*
  defines *ia*:*ia*≡(*if* ∀ *m*. *is-zero-row-upt-k m k A then 0 else to-nat* (*GREATEST*' *n*. ¬ *is-zero-row-upt-k n k A*) + *1*)
  defines *B*:*B*≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
  assumes *i-le*: *i* < *i* + *1*
  and *not-zero-i-suc-k*: ¬ *is-zero-row-upt-k i* (*Suc k*) *B*
  and *not-zero-suc-i-suc-k*: ¬ *is-zero-row-upt-k* (*i* + *1*) (*Suc k*) *B*
  and *all-zero*: ∀ *m*. *is-zero-row-upt-k m k A*
  and *Amk-notzero*: *A* $ *m* $ *from-nat k* ≠ *0*
  shows (*LEAST n*. *Gauss-Jordan-in-ij A 0* (*from-nat k*) $ *i* $ *n* ≠ *0*) < (*LEAST n*. *Gauss-Jordan-in-ij A 0* (*from-nat k*) $ (*i* + *1*) $ *n* ≠ *0*)
proof −
  have *ia2*: *ia* = *0* **using** *ia all-zero* **by** *simp*
  have *B-eq-Gauss*: *B* = *Gauss-Jordan-in-ij A 0* (*from-nat k*)
    unfolding *B Gauss-Jordan-column-k-def Let-def fst-conv snd-conv ia2*
    using *all-zero Amk-notzero least-mod-type* **unfolding** *from-nat-0* **by** *auto*
  have *i*=*0* **using** *all-zero-imp-Gauss-Jordan-column-not-zero-in-row-0*[*OF all-zero - Amk-notzero*] *not-zero-i-suc-k* **unfolding** *B ia* **.**
  moreover have *i*+*1*=*0* **using** *all-zero-imp-Gauss-Jordan-column-not-zero-in-row-0*[*OF all-zero - Amk-notzero*] *not-zero-suc-i-suc-k* **unfolding** *B ia* **.**
  ultimately show *?thesis* **using** *i-le* **by** *auto*
qed

lemma *condition-3-part-3*:
  fixes *A*::'*a*::{*field*} ^'*columns*::{*mod-type*} ^'*rows*::{*mod-type*} **and** *k*::*nat*
  defines *ia*:*ia*≡(*if* ∀ *m*. *is-zero-row-upt-k m k A then 0 else to-nat* (*GREATEST*' *n*. ¬ *is-zero-row-upt-k n k A*) + *1*)
  defines *B*:*B*≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
  assumes *rref*: *reduced-row-echelon-form-upt-k A k*
  and *i-le*: *i* < *i* + *1*
  and *not-zero-i-suc-k*: ¬ *is-zero-row-upt-k i* (*Suc k*) *B*
  and *not-zero-suc-i-suc-k*: ¬ *is-zero-row-upt-k* (*i* + *1*) (*Suc k*) *B*
  and *not-zero-m*: ¬ *is-zero-row-upt-k m k A*
  and *zero-below-greatest*: ∀ *m*≥(*GREATEST*' *n*. ¬ *is-zero-row-upt-k n k A*) + *1*. *A* $ *m* $ *from-nat k* = *0*
  shows (*LEAST n*. *A* $ *i* $ *n* ≠ *0*) < (*LEAST n*. *A* $ (*i* + *1*) $ *n* ≠ *0*)
proof −
  have *ia2*: *ia*=*to-nat* (*GREATEST*' *n*. ¬ *is-zero-row-upt-k n k A*) + *1* **unfolding** *ia* **using** *not-zero-m* **by** *presburger*
  have *B-eq-A*: *B*=*A*
    unfolding *B Gauss-Jordan-column-k-def Let-def fst-conv snd-conv ia2*
    apply *simp*
    unfolding *from-nat-to-nat-greatest* **using** *zero-below-greatest* **by** *blast*
  have *rref-suc*: *reduced-row-echelon-form-upt-k A* (*Suc k*)

proof (*rule rref-suc-if-zero-below-greatest*)
show *reduced-row-echelon-form-upt-k A k* **using** *rref* **.**
show $\neg$ ($\forall$ *a. is-zero-row-upt-k a k A*) **using** *not-zero-m* **by** *fast*
show $\forall$ *a*>*GREATEST' m.* $\neg$ *is-zero-row-upt-k m k A. is-zero-row-upt-k a* (*Suc k*) *A*
proof (*clarify*)
fix *a*::*'rows* **assume** *greatest-less-a*: (*GREATEST' m.* $\neg$ *is-zero-row-upt-k m k A*) < *a*
show *is-zero-row-upt-k a* (*Suc k*) *A*
proof (*rule is-zero-row-upt-k-suc*)
show *is-zero-row-upt-k a k A* **using** *greatest-less-a row-greater-greatest-is-zero* **by** *fast*
show *A* \$ *a* \$ *from-nat k* = *0* **using** *le-Suc*[*OF greatest-less-a*] *zero-below-greatest* **by** *fast*
qed
qed
qed
show *?thesis* **using** *rref-upt-condition3*[*OF rref-suc*] *i-le not-zero-i-suc-k not-zero-suc-i-suc-k* **unfolding** *B-eq-A* **by** *blast*
qed

lemma *condition-3-part-4*:
fixes *A*::*'a*::{*field*} ^*'columns*::{*mod-type*} ^*'rows*::{*mod-type*} **and** *k*::*nat*
defines *ia*:*ia*≡(*if* $\forall$ *m. is-zero-row-upt-k m k A then 0 else to-nat* (*GREATEST' n.* $\neg$ *is-zero-row-upt-k n k A*) + *1*)
defines *B*:*B*≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
assumes *rref*: *reduced-row-echelon-form-upt-k A k* **and** *i-le*: *i* < *i* + *1*
and *not-zero-i-suc-k*: $\neg$ *is-zero-row-upt-k i* (*Suc k*) *B*
and *not-zero-suc-i-suc-k*: $\neg$ *is-zero-row-upt-k* (*i* + *1*) (*Suc k*) *B*
and *not-zero-m*: $\neg$ *is-zero-row-upt-k m k A*
and *greatest-eq-card*: *Suc* (*to-nat* (*GREATEST' n.* $\neg$ *is-zero-row-upt-k n k A*)) = *nrows A*
shows (*LEAST n. A* \$ *i* \$ *n* $\neq$ *0*) < (*LEAST n. A* \$ (*i* + *1*) \$ *n* $\neq$ *0*)
proof –
have *ia2*: *ia*=*to-nat* (*GREATEST' n.* $\neg$ *is-zero-row-upt-k n k A*) + *1* **unfolding** *ia* **using** *not-zero-m* **by** *presburger*
have *B-eq-A*: *B*=*A*
unfolding *B Gauss-Jordan-column-k-def Let-def fst-conv snd-conv ia2*
unfolding *from-nat-to-nat-greatest* **using** *greatest-eq-card* **by** *simp*
have *greatest-eq-minus-1*: (*GREATEST' n.* $\neg$ *is-zero-row-upt-k n k A*) = -1
using *a-eq-minus-1 greatest-eq-card to-nat-plus-one-less-card* **unfolding** *nrows-def* **by** *fastforce*
have *rref-suc*: *reduced-row-echelon-form-upt-k A* (*Suc k*)
proof (*rule rref-suc-if-all-rows-not-zero*)
show *reduced-row-echelon-form-upt-k A k* **using** *rref* **.**
show $\forall$ *n.* $\neg$ *is-zero-row-upt-k n k A* **using** *Greatest-is-minus-1 greatest-eq-minus-1 greatest-ge-nonzero-row'*[*OF rref* -] *not-zero-m* **by** *metis*

qed
show *?thesis* **using** *rref-upt-condition3*[*OF rref-suc*] *i-le not-zero-i-suc-k not-zero-suc-i-suc-k*
unfolding *B-eq-A* **by** *blast*
qed

lemma *condition-3-part-5*:
fixes $A::'a::\{field\}$ ^ $'columns::\{mod\text{-}type\}$ ^ $'rows::\{mod\text{-}type\}$ **and** $k::nat$
defines *ia*:*ia*≡(*if* ∀ *m*. *is-zero-row-upt-k m k A then 0 else to-nat* (*GREATEST'* *n*. ¬ *is-zero-row-upt-k n k A*) + *1*)
defines *B*:*B*≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
assumes *rref*: *reduced-row-echelon-form-upt-k A k*
and *i-le*: *i* < *i* + *1*
and *not-zero-i-suc-k*: ¬ *is-zero-row-upt-k i* (*Suc k*) *B*
and *not-zero-suc-i-suc-k*: ¬ *is-zero-row-upt-k* (*i* + *1*) (*Suc k*) *B*
and *not-zero-m*: ¬ *is-zero-row-upt-k m k A*
and *greatest-not-card*: *Suc* (*to-nat* (*GREATEST'* *n*. ¬ *is-zero-row-upt-k n k A*)) ≠ *nrows A*
and *greatest-less-ma*: (*GREATEST'* *n*. ¬ *is-zero-row-upt-k n k A*) + *1* ≤ *ma*
and *A-ma-k-not-zero*: *A* $ *ma* $ *from-nat k* ≠ *0*
shows (*LEAST n*. *Gauss-Jordan-in-ij A* ((*GREATEST'* *n*. ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ *i* $ *n* ≠ *0*)
< (*LEAST n*. *Gauss-Jordan-in-ij A* ((*GREATEST'* *n*. ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ (*i* + *1*) $ *n* ≠ *0*)
proof −
have *ia2*: *ia*=*to-nat* (*GREATEST'* *n*. ¬ *is-zero-row-upt-k n k A*) + *1* **unfolding** *ia* **using** *not-zero-m* **by** *presburger*
have *B-eq-Gauss*: *B* = *Gauss-Jordan-in-ij A* ((*GREATEST'* *n*. ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*)
unfolding *B Gauss-Jordan-column-k-def*
unfolding *ia2 Let-def fst-conv snd-conv*
using *greatest-not-card greatest-less-ma A-ma-k-not-zero*
by (*auto simp add*: *from-nat-to-nat-greatest*)
have *suc-greatest-not-zero*: (*GREATEST'* *n*. ¬ *is-zero-row-upt-k n k A*) + *1* ≠ *0*
using *Suc-eq-plus1 suc-not-zero greatest-not-card* **unfolding** *nrows-def* **by** *auto*
show *?thesis*
proof (*cases is-zero-row-upt-k* (*i* + *1*) *k A*)
case *True*
have *zero-i-plus-one-k-B*: *is-zero-row-upt-k* (*i*+*1*) *k B*
by (*unfold B-eq-Gauss*, *rule is-zero-after-Gauss*[*OF True not-zero-m rref greatest-less-ma A-ma-k-not-zero*])
hence *Gauss-Jordan-i-not-0*: *Gauss-Jordan-in-ij A* ((*GREATEST'* *n*. ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ (*i*+*1*) $ (*from-nat k*) ≠ *0*
using *not-zero-suc-i-suc-k* **unfolding** *B-eq-Gauss* **using** *is-zero-row-upt-k-suc* **by** *blast*
have *i-plus-one-eq*: *i* + *1* = ((*GREATEST'* *n*. ¬ *is-zero-row-upt-k n k A*) + *1*)
proof (*rule ccontr*)
assume *i-not-greatest*: *i* + *1* ≠ (*GREATEST'* *n*. ¬ *is-zero-row-upt-k n k A*)

+ *1*
have *Gauss-Jordan-in-ij A* ((*GREATEST'* *n*. ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) \$ (*i* + *1*) \$ (*from-nat k*) = *0*
proof (*rule Gauss-Jordan-in-ij-0*)
show ∃ *n*. *A* \$ *n* \$ *from-nat k* ≠ *0* ∧ (*GREATEST'* *n*. ¬ *is-zero-row-upt-k n k A*) + *1* ≤ *n* **using** *greatest-less-ma A-ma-k-not-zero* **by** *blast*
show *i* + *1* ≠ (*GREATEST'* *n*. ¬ *is-zero-row-upt-k n k A*) + *1* **using** *i-not-greatest* **.**
qed
thus *False* **using** *Gauss-Jordan-i-not-0* **by** *contradiction*
qed
hence *i-eq-greatest*: *i*=(*GREATEST'* *n*. ¬ *is-zero-row-upt-k n k A*) **using** *add-right-cancel* **by** *simp*
have *Least-eq-k*: (*LEAST ka*. *Gauss-Jordan-in-ij A* ((*GREATEST'* *n*. ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) \$ (*i*+*1*) \$ *ka* ≠ *0*) = *from-nat k*
proof (*rule Least-equality*)
show *Gauss-Jordan-in-ij A* ((*GREATEST'* *n*. ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) \$ (*i*+*1*) \$ *from-nat k* ≠ *0* **by** (*metis Gauss-Jordan-i-not-0*)
fix *y* **assume** *Gauss-Jordan-in-ij A* ((*GREATEST'* *n*. ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) \$ (*i*+*1*) \$ *y* ≠ *0*
thus *from-nat k* ≤ *y* **using** *zero-i-plus-one-k-B* **unfolding** *i-eq-greatest B-eq-Gauss* **by** (*metis is-zero-row-upt-k-def not-less to-nat-le*)
qed
have *not-zero-i-A*: ¬ *is-zero-row-upt-k i k A* **using** *greatest-less-zero-row*[*OF rref*] *not-zero-m* **unfolding** *i-eq-greatest* **by** *fast*
from *this* **obtain** *j* **where** *Aij-not-0*: *A* \$ *i* \$ *j* ≠ *0* **and** *j-le-k*: *to-nat j* < *k* **unfolding** *is-zero-row-upt-k-def* **by** *blast*
have *least-le-k*: *to-nat* (*LEAST ka*. *A* \$ *i* \$ *ka* ≠ *0*) < *k*
by (*metis* (*lifting*, *mono-tags*) *Aij-not-0 j-le-k less-trans linorder-cases not-less-Least to-nat-mono*)
have *Least-eq*: (*LEAST n*. *Gauss-Jordan-in-ij A* ((*GREATEST'* *n*. ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) \$ *i* \$ *n* ≠ *0*) =
(*LEAST n*. *A* \$ *i* \$ *n* ≠ *0*)
proof (*rule Least-equality*)
show *Gauss-Jordan-in-ij A* ((*GREATEST'* *n*. ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) \$ *i* \$ (*LEAST ka*. *A* \$ *i* \$ *ka* ≠ *0*) ≠ *0*
using *Gauss-Jordan-in-ij-preserves-previous-elements*[*OF rref not-zero-i-A - suc-greatest-not-zero least-le-k*] *greatest-less-ma A-ma-k-not-zero*
using *rref-upt-condition2*[*OF rref*] *not-zero-i-A* **by** *fastforce*
fix *y* **assume** *Gauss-Jordan-y*:*Gauss-Jordan-in-ij A* ((*GREATEST'* *n*. ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) \$ *i* \$ *y* ≠ *0*
show (*LEAST ka*. *A* \$ *i* \$ *ka* ≠ *0*) ≤ *y*
proof (*cases to-nat y* < *k*)
case *False* **thus** *?thesis* **by** (*metis dual-linorder.not-le least-le-k less-trans to-nat-mono*)
next
case *True*
have *A* \$ *i* \$ *y* ≠ *0* **using** *Gauss-Jordan-y* **using** *Gauss-Jordan-in-ij-preserves-previous-elements*[*OF rref not-zero-m - suc-greatest-not-zero True*]

using *A-ma-k-not-zero greatest-less-ma* **by** *fastforce*
thus *?thesis* **using** *Least-le* **by** *fastforce*
qed
qed
also have ... < *from-nat k* **by** (*metis is-zero-row-upt-k-def is-zero-row-upt-k-suc le-less-linear le-less-trans least-le-k not-zero-suc-i-suc-k to-nat-mono′ zero-i-plus-one-k-B*)

finally show *?thesis* **unfolding** *Least-eq-k* **.**
next
case *False*
have *not-zero-i-A*: ¬ *is-zero-row-upt-k i k A* **using** *rref-upt-condition1*[*OF rref*] *False i-le* **by** *blast*
from *this* **obtain** *j* **where** *Aij-not-0*: *A* \$ *i* \$ *j* ≠ *0* **and** *j-le-k*: *to-nat j* < *k* **unfolding** *is-zero-row-upt-k-def* **by** *blast*
have *least-le-k*: *to-nat* (*LEAST ka. A* \$ *i* \$ *ka* ≠ *0*) < *k*
by (*metis* (*lifting*, *mono-tags*) *Aij-not-0 j-le-k less-trans linorder-cases not-less-Least to-nat-mono*)
have *Least-i-eq*: (*LEAST n. Gauss-Jordan-in-ij A* ((*GREATEST′ n.* ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) \$ *i* \$ *n* ≠ *0*)
= (*LEAST n. A* \$ *i* \$ *n* ≠ *0*)
proof (*rule Least-equality*)
show *Gauss-Jordan-in-ij A* ((*GREATEST′ n.* ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) \$ *i* \$ (*LEAST ka. A* \$ *i* \$ *ka* ≠ *0*) ≠ *0*
using *Gauss-Jordan-in-ij-preserves-previous-elements*[*OF rref not-zero-i-A - suc-greatest-not-zero least-le-k*] *greatest-less-ma A-ma-k-not-zero*
using *rref-upt-condition2*[*OF rref*] *not-zero-i-A* **by** *fastforce*
fix *y* **assume** *Gauss-Jordan-y*:*Gauss-Jordan-in-ij A* ((*GREATEST′ n.* ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) \$ *i* \$ *y* ≠ *0*
show (*LEAST ka. A* \$ *i* \$ *ka* ≠ *0*) ≤ *y*
proof (*cases to-nat y* < *k*)
case *False* **thus** *?thesis* **by** (*metis dual-linorder.not-le dual-linorder.not-less-iff-gr-or-eq le-less-trans least-le-k to-nat-mono*)
next
case *True*
have *A* \$ *i* \$ *y* ≠ *0* **using** *Gauss-Jordan-y* **using** *Gauss-Jordan-in-ij-preserves-previous-elements*[*OF rref not-zero-m - suc-greatest-not-zero True*]
using *A-ma-k-not-zero greatest-less-ma* **by** *fastforce*
thus *?thesis* **using** *Least-le* **by** *fastforce*
qed
qed
from *False* **obtain** *s* **where** *Ais-not-0*: *A* \$ (*i+1*) \$ *s* ≠ *0* **and** *s-le-k*: *to-nat s* < *k* **unfolding** *is-zero-row-upt-k-def* **by** *blast*
have *least-le-k*: *to-nat* (*LEAST ka. A* \$ (*i+1*) \$ *ka* ≠ *0*) < *k*
by (*metis* (*lifting*, *mono-tags*) *Ais-not-0 s-le-k dual-linorder.neq-iff less-trans not-less-Least to-nat-mono*)
have *Least-i-plus-one-eq*: (*LEAST n. Gauss-Jordan-in-ij A* ((*GREATEST′ n.* ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) \$ (*i+1*) \$ *n* ≠ *0*)
= (*LEAST n. A* \$ (*i+1*) \$ *n* ≠ *0*)
proof (*rule Least-equality*)

show *Gauss-Jordan-in-ij A* ((*GREATEST′ n*. ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ (*i*+*1*) $ (*LEAST ka*. *A* $ (*i*+*1*) $ *ka* ≠ *0*) ≠ *0*
using *Gauss-Jordan-in-ij-preserves-previous-elements*[*OF rref not-zero-i-A - suc-greatest-not-zero least-le-k*] *greatest-less-ma A-ma-k-not-zero*
using *rref-upt-condition2*[*OF rref*] *False* **by** *fastforce*
fix *y* **assume** *Gauss-Jordan-y*:*Gauss-Jordan-in-ij A* ((*GREATEST′ n*. ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ (*i*+*1*) $ *y* ≠ *0*
show (*LEAST ka*. *A* $ (*i*+*1*) $ *ka* ≠ *0*) ≤ *y*
proof (*cases to-nat y* < *k*)
case *False* **thus** *?thesis* **by** (*metis* (*mono-tags*) *dual-linorder.le-less-linear least-le-k less-trans to-nat-mono*)
next
case *True*
have *A* $ (*i*+*1*) $ *y* ≠ *0* **using** *Gauss-Jordan-y* **using** *Gauss-Jordan-in-ij-preserves-previous-elements*[*OF rref not-zero-m - suc-greatest-not-zero True*]
using *A-ma-k-not-zero greatest-less-ma* **by** *fastforce*
thus *?thesis* **using** *Least-le* **by** *fastforce*
qed
qed
show *?thesis* **unfolding** *Least-i-plus-one-eq Least-i-eq* **using** *rref-upt-condition3*[*OF rref*] *i-le False not-zero-i-A* **by** *blast*
qed
qed

lemma *condition-3*:
fixes *A*::′*a*::{*field*} ^′*columns*::{*mod-type*} ^′*rows*::{*mod-type*} **and** *k*::*nat*
defines *ia*:*ia*≡(*if* ∀ *m*. *is-zero-row-upt-k m k A then 0 else to-nat* (*GREATEST′ n*. ¬ *is-zero-row-upt-k n k A*) + *1*)
defines *B*:*B*≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
assumes *rref*: *reduced-row-echelon-form-upt-k A k*
and *i-le*: *i* < *i* + *1*
and *not-zero-i-suc-k*: ¬ *is-zero-row-upt-k i* (*Suc k*) *B*
and *not-zero-suc-i-suc-k*: ¬ *is-zero-row-upt-k* (*i* + *1*) (*Suc k*) *B*
shows (*LEAST n*. *B* $ *i* $ *n* ≠ *0*) < (*LEAST n*. *B* $ (*i* + *1*) $ *n* ≠ *0*)
proof (*unfold B Gauss-Jordan-column-k-def ia Let-def fst-conv snd-conv*, *auto*, *unfold from-nat-to-nat-greatest from-nat-0*)
assume *all-zero*: ∀ *m*. *is-zero-row-upt-k m k A*
and *all-zero-k*: ∀ *m*≥*0*. *A* $ *m* $ *from-nat k* = *0*
show (*LEAST n*. *A* $ *i* $ *n* ≠ *0*) < (*LEAST n*. *A* $ (*i* + *1*) $ *n* ≠ *0*)
using *condition-3-part-1*[*OF - all-zero*] **using** *all-zero-k least-mod-type not-zero-i-suc-k* **unfolding** *B ia* **by** *fast*
next
fix *m* **assume** *all-zero*: ∀ *m*. *is-zero-row-upt-k m k A*
and *Amk-notzero*: *A* $ *m* $ *from-nat k* ≠ *0*
show (*LEAST n*. *Gauss-Jordan-in-ij A 0* (*from-nat k*) $ *i* $ *n* ≠ *0*) < (*LEAST n*. *Gauss-Jordan-in-ij A 0* (*from-nat k*) $ (*i* + *1*) $ *n* ≠ *0*)
using *condition-3-part-2*[*OF i-le - - all-zero Amk-notzero*] **using** *not-zero-i-suc-k not-zero-suc-i-suc-k* **unfolding** *B ia* **.**
next

fix *m*
assume *not-zero-m*: $\neg$ *is-zero-row-upt-k m k A*
and *zero-below-greatest*: $\forall m \geq (GREATEST'\ n.\ \neg$ *is-zero-row-upt-k n k A*$) + 1.\ A\ \$\ m\ \$$ *from-nat* $k = 0$
show $(LEAST\ n.\ A\ \$\ i\ \$\ n \neq 0) < (LEAST\ n.\ A\ \$\ (i + 1)\ \$\ n \neq 0)$
using *condition-3-part-3*[*OF rref i-le - - not-zero-m zero-below-greatest*] **using** *not-zero-i-suc-k not-zero-suc-i-suc-k* **unfolding** *B ia* **.**
next
fix *m*
assume *not-zero-m*: $\neg$ *is-zero-row-upt-k m k A*
and *greatest-eq-card*: *Suc* (*to-nat* ($GREATEST'\ n.\ \neg$ *is-zero-row-upt-k n k A*)) = *nrows A*
show $(LEAST\ n.\ A\ \$\ i\ \$\ n \neq 0) < (LEAST\ n.\ A\ \$\ (i + 1)\ \$\ n \neq 0)$
using *condition-3-part-4*[*OF rref i-le - - not-zero-m greatest-eq-card*] **using** *not-zero-i-suc-k not-zero-suc-i-suc-k* **unfolding** *B ia* **.**
next
fix *m ma*
assume *not-zero-m*: $\neg$ *is-zero-row-upt-k m k A*
and *greatest-not-card*: *Suc* (*to-nat* ($GREATEST'\ n.\ \neg$ *is-zero-row-upt-k n k A*)) $\neq$ *nrows A*
and *greatest-less-ma*: ($GREATEST'\ n.\ \neg$ *is-zero-row-upt-k n k A*) + *1* $\leq$ *ma*
and *A-ma-k-not-zero*: $A\ \$\ ma\ \$$ *from-nat* $k \neq 0$
show (*LEAST n. Gauss-Jordan-in-ij A* (($GREATEST'\ n.\ \neg$ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $\$\ i\ \$\ n \neq 0$)
< (*LEAST n. Gauss-Jordan-in-ij A* (($GREATEST'\ n.\ \neg$ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $\$\ (i + 1)\ \$\ n \neq 0$)
using *condition-3-part-5*[*OF rref i-le - - not-zero-m greatest-not-card greatest-less-ma A-ma-k-not-zero*]
using *not-zero-i-suc-k not-zero-suc-i-suc-k* **unfolding** *B ia* **.**
qed

lemma *condition-4-part-1*:
fixes *A*::*'a*::{*field*} ^*'columns*::{*mod-type*} ^*'rows*::{*mod-type*} **and** *k*::*nat*
defines *ia*:*ia*≡(*if* $\forall m.$ *is-zero-row-upt-k m k A then 0 else to-nat* ($GREATEST'\ n.\ \neg$ *is-zero-row-upt-k n k A*) + *1*)
defines *B*:*B*≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
assumes *not-zero-i-suc-k*: $\neg$ *is-zero-row-upt-k i* (*Suc k*) *B*
and *all-zero*: $\forall m.$ *is-zero-row-upt-k m k A*
and *all-zero-k*: $\forall m.\ A\ \$\ m\ \$$ *from-nat* $k = 0$
shows $A\ \$\ j\ \$\ (LEAST\ n.\ A\ \$\ i\ \$\ n \neq 0) = 0$
proof –
have *ia2*: *ia* = *0* **using** *ia all-zero* **by** *simp*
have *B-eq-A*: *B*=*A* **unfolding** *B Gauss-Jordan-column-k-def Let-def fst-conv snd-conv ia2* **using** *all-zero-k* **by** *fastforce*
show *?thesis* **using** *B-eq-A all-zero all-zero-k is-zero-row-upt-k-suc not-zero-i-suc-k* **by** *blast*
qed

lemma *condition-4-part-2*:
fixes *A*::*'a*::{*field*} ^*'columns*::{*mod-type*} ^*'rows*::{*mod-type*} **and** *k*::*nat*
defines *ia*:*ia*≡(*if* ∀ *m*. *is-zero-row-upt-k m k A then 0 else to-nat* (*GREATEST' n*. ¬ *is-zero-row-upt-k n k A*) + *1*)
defines *B*:*B*≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
assumes *not-zero-i-suc-k*: ¬ *is-zero-row-upt-k i* (*Suc k*) *B*
and *i-not-j*: *i* ≠ *j*
and *all-zero*: ∀ *m*. *is-zero-row-upt-k m k A*
and *Amk-not-zero*: *A* $ *m* $ *from-nat k* ≠ *0*
shows *Gauss-Jordan-in-ij A 0* (*from-nat k*) $ *j* $ (*LEAST n*. *Gauss-Jordan-in-ij A 0* (*from-nat k*) $ *i* $ *n* ≠ *0*) = *0*
proof −
have *i-eq-0*: *i*=*0* **using** *all-zero-imp-Gauss-Jordan-column-not-zero-in-row-0*[*OF all-zero - Amk-not-zero*] *not-zero-i-suc-k* **unfolding** *B ia* **.**
have *least-eq-k*: (*LEAST n*. *Gauss-Jordan-in-ij A 0* (*from-nat k*) $ *i* $ *n* ≠ *0*) = *from-nat k*
proof (*rule Least-equality*)
show *Gauss-Jordan-in-ij A 0* (*from-nat k*) $ *i* $ *from-nat k* ≠ *0* **unfolding** *i-eq-0* **using** *Amk-not-zero Gauss-Jordan-in-ij-1 least-mod-type zero-neq-one* **by** *fastforce*
fix *y* **assume** *Gauss-Jordan-y-not-0*: *Gauss-Jordan-in-ij A 0* (*from-nat k*) $ *i* $ *y* ≠ *0*
show *from-nat k* ≤ *y*
proof (*rule ccontr*)
assume ¬ *from-nat k* ≤ *y*
hence *y* < (*from-nat k*) **by** *simp*
hence *to-nat-y-less-k*: *to-nat y* < *k* **using** *to-nat-le* **by** *auto*
have *Gauss-Jordan-in-ij A 0* (*from-nat k*) $ *i* $ *y* = *0*
using *Gauss-Jordan-in-ij-preserves-previous-elements'*[*OF all-zero to-nat-y-less-k Amk-not-zero*] *all-zero to-nat-y-less-k*
unfolding *is-zero-row-upt-k-def* **by** *fastforce*
thus *False* **using** *Gauss-Jordan-y-not-0* **by** *contradiction*
qed
qed
show *?thesis* **unfolding** *least-eq-k* **apply** (*rule Gauss-Jordan-in-ij-0*) **using** *i-eq-0 i-not-j Amk-not-zero least-mod-type* **by** *blast+*
qed

lemma *condition-4-part-3*:
fixes *A*::*'a*::{*field*} ^*'columns*::{*mod-type*} ^*'rows*::{*mod-type*} **and** *k*::*nat*
defines *ia*:*ia*≡(*if* ∀ *m*. *is-zero-row-upt-k m k A then 0 else to-nat* (*GREATEST' n*. ¬ *is-zero-row-upt-k n k A*) + *1*)
defines *B*:*B*≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
assumes *rref*: *reduced-row-echelon-form-upt-k A k*
and *not-zero-i-suc-k*: ¬ *is-zero-row-upt-k i* (*Suc k*) *B*
and *i-not-j*: *i* ≠ *j*
and *not-zero-m*: ¬ *is-zero-row-upt-k m k A*

and *zero-below-greatest*: $\forall\, m \geq (GREATEST'\ n.\ \neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ n\ k\ A) + 1.$ $A\ \$\ m\ \$\ from\text{-}nat\ k = 0$
shows $A\ \$\ j\ \$\ (LEAST\ n.\ A\ \$\ i\ \$\ n \neq 0) = 0$
proof –
have *ia2*: *ia=to-nat* (*GREATEST' n.* ¬ *is-zero-row-upt-k n k A*) + *1* **unfolding** *ia* **using** *not-zero-m* **by** *presburger*
have *B-eq-A*: *B=A*
unfolding *B Gauss-Jordan-column-k-def Let-def fst-conv snd-conv ia2*
apply *simp*
unfolding *from-nat-to-nat-greatest* **using** *zero-below-greatest* **by** *blast*
have *rref-suc*: *reduced-row-echelon-form-upt-k A* (*Suc k*)
proof (*rule rref-suc-if-zero-below-greatest*[*OF rref*], *auto intro*!: *not-zero-m*)
fix *a*
assume *greatest-less-a*: (*GREATEST' m.* ¬ *is-zero-row-upt-k m k A*) < *a*
show *is-zero-row-upt-k a* (*Suc k*) *A*
proof (*rule is-zero-row-upt-k-suc*)
show *is-zero-row-upt-k a k A* **using** *row-greater-greatest-is-zero*[*OF greatest-less-a*]
.
show *A* \$ *a* \$ *from-nat k* = *0* **using** *zero-below-greatest le-Suc*[*OF greatest-less-a*] **by** *blast*
qed
qed
show *?thesis* **using** *rref-upt-condition4*[*OF rref-suc*] *not-zero-i-suc-k i-not-j* **unfolding** *B-eq-A* **by** *blast*
qed

lemma *condition-4-part-4*:
fixes *A*::*'a*::{*field*} ^*'columns*::{*mod-type*} ^*'rows*::{*mod-type*} **and** *k*::*nat*
defines *ia*:*ia*≡(*if* ∀ *m. is-zero-row-upt-k m k A then 0 else to-nat* (*GREATEST' n.* ¬ *is-zero-row-upt-k n k A*) + *1*)
defines *B*:*B*≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
assumes *rref*: *reduced-row-echelon-form-upt-k A k*
and *not-zero-i-suc-k*: ¬ *is-zero-row-upt-k i* (*Suc k*) *B*
and *i-not-j*: $i \neq j$
and *not-zero-m*: ¬ *is-zero-row-upt-k m k A*
and *greatest-eq-card*: *Suc* (*to-nat* (*GREATEST' n.* ¬ *is-zero-row-upt-k n k A*)) = *nrows A*
shows $A\ \$\ j\ \$\ (LEAST\ n.\ A\ \$\ i\ \$\ n \neq 0) = 0$
proof –
have *ia2*: *ia=to-nat* (*GREATEST' n.* ¬ *is-zero-row-upt-k n k A*) + *1* **unfolding** *ia* **using** *not-zero-m* **by** *presburger*
have *B-eq-A*: *B=A*
unfolding *B Gauss-Jordan-column-k-def Let-def fst-conv snd-conv ia2*
unfolding *from-nat-to-nat-greatest* **using** *greatest-eq-card* **by** *simp*
have *greatest-eq-minus-1*: (*GREATEST' n.* ¬ *is-zero-row-upt-k n k A*) = −*1*
using *a-eq-minus-1 greatest-eq-card to-nat-plus-one-less-card* **unfolding** *nrows-def* **by** *fastforce*
have *rref-suc*: *reduced-row-echelon-form-upt-k A* (*Suc k*)
proof (*rule rref-suc-if-all-rows-not-zero*)

show *reduced-row-echelon-form-upt-k A k* **using** *rref* **.**
show ∀ *n*. ¬ *is-zero-row-upt-k n k A* **using** *Greatest-is-minus-1 greatest-eq-minus-1 greatest-ge-nonzero-row'*[*OF rref* -] *not-zero-m* **by** *metis*
qed
show *?thesis* **using** *rref-upt-condition4*[*OF rref-suc*] **using** *not-zero-i-suc-k i-not-j* **unfolding** *B-eq-A i-not-j* **by** *blast*
qed

lemma *condition-4-part-5*:
fixes *A*::*'a*::{*field*} ^*'columns*::{*mod-type*} ^*'rows*::{*mod-type*} **and** *k*::*nat*
defines *ia*:*ia*≡(*if* ∀ *m*. *is-zero-row-upt-k m k A then 0 else to-nat* (*GREATEST' n*. ¬ *is-zero-row-upt-k n k A*) + *1*)
defines *B*:*B*≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
assumes *rref*: *reduced-row-echelon-form-upt-k A k*
and *not-zero-i-suc-k*: ¬ *is-zero-row-upt-k i* (*Suc k*) *B*
and *i-not-j*: *i* ≠ *j*
and *not-zero-m*: ¬ *is-zero-row-upt-k m k A*
and *greatest-not-card*: *Suc* (*to-nat* (*GREATEST' n*. ¬ *is-zero-row-upt-k n k A*)) ≠ *nrows A*
and *greatest-less-ma*: (*GREATEST' n*. ¬ *is-zero-row-upt-k n k A*) + *1* ≤ *ma*
and *A-ma-k-not-zero*: *A* $ *ma* $ *from-nat k* ≠ *0*
shows *Gauss-Jordan-in-ij A* ((*GREATEST' n*. ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ *j* $
(*LEAST n*. *Gauss-Jordan-in-ij A* ((*GREATEST' n*. ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ *i* $ *n* ≠ *0*) = *0*
proof −
have *ia2*: *ia*=*to-nat* (*GREATEST' n*. ¬ *is-zero-row-upt-k n k A*) + *1* **unfolding** *ia* **using** *not-zero-m* **by** *presburger*
have *B-eq-Gauss*: *B* = *Gauss-Jordan-in-ij A* ((*GREATEST' n*. ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*)
unfolding *B Gauss-Jordan-column-k-def*
unfolding *ia2 Let-def fst-conv snd-conv*
using *greatest-not-card greatest-less-ma A-ma-k-not-zero*
by (*auto simp add*: *from-nat-to-nat-greatest*)
have *suc-greatest-not-zero*: (*GREATEST' n*. ¬ *is-zero-row-upt-k n k A*) + *1* ≠ *0*
using *Suc-eq-plus1 suc-not-zero greatest-not-card* **unfolding** *nrows-def* **by** *auto*
show *?thesis*
proof (*cases is-zero-row-upt-k i k A*)
case *True*
have *zero-i-k-B*: *is-zero-row-upt-k i k B* **unfolding** *B-eq-Gauss* **by** (*rule is-zero-after-Gauss*[*OF True not-zero-m rref greatest-less-ma A-ma-k-not-zero*])
hence *Gauss-Jordan-i-not-0*: *Gauss-Jordan-in-ij A* ((*GREATEST' n*. ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ (*i*) $ (*from-nat k*) ≠ *0*
using *not-zero-i-suc-k* **unfolding** *B-eq-Gauss* **using** *is-zero-row-upt-k-suc* **by** *blast*
have *i-eq-greatest*: *i* = ((*GREATEST' n*. ¬ *is-zero-row-upt-k n k A*) + *1*)
proof (*rule ccontr*)
assume *i-not-greatest*: *i* ≠ (*GREATEST' n*. ¬ *is-zero-row-upt-k n k A*) + *1*

have *Gauss-Jordan-in-ij A* ((*GREATEST'* *n*. ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) \$ *i* \$ (*from-nat k*) = *0*
proof (*rule Gauss-Jordan-in-ij-0*)
show ∃ *n*. *A* \$ *n* \$ *from-nat k* ≠ *0* ∧ (*GREATEST'* *n*. ¬ *is-zero-row-upt-k n k A*) + *1* ≤ *n* **using** *greatest-less-ma A-ma-k-not-zero* **by** *blast*
show *i* ≠ (*GREATEST'* *n*. ¬ *is-zero-row-upt-k n k A*) + *1* **using** *i-not-greatest* **.**
qed
thus *False* **using** *Gauss-Jordan-i-not-0* **by** *contradiction*
qed
have *Gauss-Jordan-i-1*: *Gauss-Jordan-in-ij A* ((*GREATEST'* *n*. ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) \$ *i* \$ (*from-nat k*) = *1*
unfolding *i-eq-greatest* **using** *Gauss-Jordan-in-ij-1 greatest-less-ma A-ma-k-not-zero* **by** *blast*
have *Least-eq-k*: (*LEAST ka*. *Gauss-Jordan-in-ij A* ((*GREATEST'* *n*. ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) \$ *i* \$ *ka* ≠ *0*) = *from-nat k*
proof (*rule Least-equality*)
show *Gauss-Jordan-in-ij A* ((*GREATEST'* *n*. ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) \$ *i* \$ *from-nat k* ≠ *0* **using** *Gauss-Jordan-i-not-0* **.**
fix *y* **assume** *Gauss-Jordan-in-ij A* ((*GREATEST'* *n*. ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) \$ *i* \$ *y* ≠ *0*
thus *from-nat k* ≤ *y* **using** *zero-i-k-B* **unfolding** *i-eq-greatest B-eq-Gauss* **by** (*metis is-zero-row-upt-k-def not-less to-nat-le*)
qed
show *?thesis* **using** *A-ma-k-not-zero Gauss-Jordan-in-ij-0' Least-eq-k greatest-less-ma i-eq-greatest i-not-j* **by** *force*
next
case *False*
obtain *n* **where** *Ain-not-0*: *A* \$ *i* \$ *n* ≠ *0* **and** *j-le-k*: *to-nat n* < *k* **using** *False* **unfolding** *is-zero-row-upt-k-def* **by** *auto*
have *least-le-k*: *to-nat* (*LEAST ka*. *A* \$ *i* \$ *ka* ≠ *0*) < *k*
by (*metis* (*lifting*, *mono-tags*) *Ain-not-0 dual-linorder.neq-iff j-le-k less-trans not-less-Least to-nat-mono*)
have *Least-eq*: (*LEAST ka*. *Gauss-Jordan-in-ij A* ((*GREATEST'* *n*. ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) \$ *i* \$ *ka* ≠ *0*)
= (*LEAST ka*. *A* \$ *i* \$ *ka* ≠ *0*)
proof (*rule Least-equality*)
show *Gauss-Jordan-in-ij A* ((*GREATEST'* *n*. ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) \$ *i* \$ (*LEAST ka*. *A* \$ *i* \$ *ka* ≠ *0*) ≠ *0*
using *Gauss-Jordan-in-ij-preserves-previous-elements*[*OF rref False - suc-greatest-not-zero least-le-k*] **using** *greatest-less-ma A-ma-k-not-zero*
using *rref-upt-condition2*[*OF rref*] *False* **by** *fastforce*
fix *y* **assume** *Gauss-Jordan-y*:*Gauss-Jordan-in-ij A* ((*GREATEST'* *n*. ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) \$ *i* \$ *y* ≠ *0*
show (*LEAST ka*. *A* \$ *i* \$ *ka* ≠ *0*) ≤ *y*
proof (*cases to-nat y* < *k*)
case *False* **show** *?thesis* **by** (*metis* (*mono-tags*) *False least-le-k less-trans not-leE to-nat-from-nat to-nat-le*)
next

case *True*
have *A $ i $ y* $\neq$ *0*
using *Gauss-Jordan-y* **using** *Gauss-Jordan-in-ij-preserves-previous-elements*[*OF rref not-zero-m - suc-greatest-not-zero True*]
using *A-ma-k-not-zero greatest-less-ma* **by** *fastforce*
thus *?thesis* **by** (*rule Least-le*)
qed
qed
have *Gauss-Jordan-eq-A*: *Gauss-Jordan-in-ij A* ((*GREATEST' n.* ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) *$ j $* (*LEAST n. A $ i $ n* $\neq$ *0*) =
A $ j $ (*LEAST n. A $ i $ n* $\neq$ *0*)
using *Gauss-Jordan-in-ij-preserves-previous-elements*[*OF rref not-zero-m - suc-greatest-not-zero least-le-k*]
using *A-ma-k-not-zero greatest-less-ma* **by** *fastforce*
show *?thesis* **unfolding** *Least-eq* **using** *rref-upt-condition4*[*OF rref*]
using *False Gauss-Jordan-eq-A i-not-j* **by** *presburger*
qed
qed

lemma *condition-4*:
fixes *A::'a::{field}^'columns::{mod-type}^'rows::{mod-type}* **and** *k::nat*
defines *ia:ia≡(if ∀ m. is-zero-row-upt-k m k A then 0 else to-nat* (*GREATEST' n.* ¬ *is-zero-row-upt-k n k A*) + *1*)
defines *B:B≡(snd* (*Gauss-Jordan-column-k* (*ia,A*) *k*))
assumes *rref*: *reduced-row-echelon-form-upt-k A k*
and *not-zero-i-suc-k*: ¬ *is-zero-row-upt-k i* (*Suc k*) *B*
and *i-not-j*: *i* $\neq$ *j*
shows *B $ j $* (*LEAST n. B $ i $ n* $\neq$ *0*) = *0*
proof (*unfold B Gauss-Jordan-column-k-def ia Let-def fst-conv snd-conv*, *auto*, *unfold from-nat-to-nat-greatest from-nat-0*)
assume *all-zero*: *∀ m. is-zero-row-upt-k m k A*
and *all-zero-k*: *∀ m≥0. A $ m $ from-nat k* = *0*
show *A $ j $* (*LEAST n. A $ i $ n* $\neq$ *0*) = *0* **using** *condition-4-part-1*[*OF - all-zero*] **using** *all-zero-k not-zero-i-suc-k least-mod-type* **unfolding** *B ia* **by** *blast*
next
fix *m*
assume *all-zero*: *∀ m. is-zero-row-upt-k m k A*
and *Amk-not-zero*: *A $ m $ from-nat k* $\neq$ *0*
show *Gauss-Jordan-in-ij A 0* (*from-nat k*) *$ j $* (*LEAST n. Gauss-Jordan-in-ij A 0* (*from-nat k*) *$ i $ n* $\neq$ *0*) = *0*
using *condition-4-part-2*[*OF - i-not-j all-zero Amk-not-zero*] **using** *not-zero-i-suc-k* **unfolding** *B ia* **.**
next
fix *m* **assume** *not-zero-m*: ¬ *is-zero-row-upt-k m k A*
and *zero-below-greatest*: *∀ m≥*(*GREATEST' n.* ¬ *is-zero-row-upt-k n k A*) + *1*. *A $ m $ from-nat k* = *0*
show *A $ j $* (*LEAST n. A $ i $ n* $\neq$ *0*) = *0*
using *condition-4-part-3*[*OF rref - i-not-j not-zero-m zero-below-greatest*] **using**

not-zero-i-suc-k **unfolding** *B ia* **.**
next
fix *m*
assume *not-zero-m*: ¬ *is-zero-row-upt-k m k A*
and *greatest-eq-card*: *Suc* (*to-nat* (*GREATEST′ n*. ¬ *is-zero-row-upt-k n k A*)) = *nrows A*
show *A* $ *j* $ (*LEAST n*. *A* $ *i* $ *n* ≠ *0*) = *0*
using *condition-4-part-4*[*OF rref - i-not-j not-zero-m greatest-eq-card*] **using** *not-zero-i-suc-k* **unfolding** *B ia* **.**
next
fix *m ma*
assume *not-zero-m*: ¬ *is-zero-row-upt-k m k A*
and *greatest-not-card*: *Suc* (*to-nat* (*GREATEST′ n*. ¬ *is-zero-row-upt-k n k A*)) ≠ *nrows A*
and *greatest-less-ma*: (*GREATEST′ n*. ¬ *is-zero-row-upt-k n k A*) + *1* ≤ *ma*
and *A-ma-k-not-zero*: *A* $ *ma* $ *from-nat k* ≠ *0*
show *Gauss-Jordan-in-ij A* ((*GREATEST′ n*. ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ *j* $
(*LEAST n*. *Gauss-Jordan-in-ij A* ((*GREATEST′ n*. ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ *i* $ *n* ≠ *0*) = *0*
using *condition-4-part-5*[*OF rref - i-not-j not-zero-m greatest-not-card greatest-less-ma A-ma-k-not-zero*] **using** *not-zero-i-suc-k* **unfolding** *B ia* **.**
qed

lemma *reduced-row-echelon-form-upt-k-Gauss-Jordan-column-k*:
fixes *A*::*′a*::{*field*} ^*′columns*::{*mod-type*} ^*′rows*::{*mod-type*} **and** *k*::*nat*
defines *ia*:*ia*≡(*if* ∀ *m*. *is-zero-row-upt-k m k A then 0 else to-nat* (*GREATEST′ n*. ¬ *is-zero-row-upt-k n k A*) + *1*)
defines *B*:*B*≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
assumes *rref*: *reduced-row-echelon-form-upt-k A k*
shows *reduced-row-echelon-form-upt-k B* (*Suc k*)
proof (*rule reduced-row-echelon-form-upt-k-intro*, *auto*)
show ⋀*i j*. *is-zero-row-upt-k i* (*Suc k*) *B* ⟹ *i* < *j* ⟹ *is-zero-row-upt-k j* (*Suc k*) *B* **using** *condition-1 assms* **by** *blast*
show ⋀*i*. ¬ *is-zero-row-upt-k i* (*Suc k*) *B* ⟹ *B* $ *i* $ (*LEAST k*. *B* $ *i* $ *k* ≠ *0*) = *1* **using** *condition-2 assms* **by** *blast*
show ⋀*i*. *i* < *i* + *1* ⟹ ¬ *is-zero-row-upt-k i* (*Suc k*) *B* ⟹ ¬ *is-zero-row-upt-k* (*i* + *1*) (*Suc k*) *B* ⟹ (*LEAST n*. *B* $ *i* $ *n* ≠ *0*) < (*LEAST n*. *B* $ (*i* + *1*) $ *n* ≠ *0*) **using** *condition-3 assms* **by** *blast*
show ⋀*i j*. ¬ *is-zero-row-upt-k i* (*Suc k*) *B* ⟹ *i* ≠ *j* ⟹ *B* $ *j* $ (*LEAST n*. *B* $ *i* $ *n* ≠ *0*) = *0* **using** *condition-4 assms* **by** *blast*
qed

lemma *foldl-Gauss-condition-1*:

fixes *A::'a::{field} ^'columns::{mod-type} ^'rows::{mod-type}* **and** *k::nat*
assumes $\forall$ *m. is-zero-row-upt-k m k A*
and $\forall$ *m≥0. A \$ m \$ from-nat k = 0*
shows *is-zero-row-upt-k m (Suc k) A*
by *(rule is-zero-row-upt-k-suc, auto simp add: assms least-mod-type)*

lemma *foldl-Gauss-condition-2*:
fixes *A::'a::{field} ^'columns::{mod-type} ^'rows::{mod-type}* **and** *k::nat*
assumes *k: k < ncols A*
and *all-zero:* $\forall$ *m. is-zero-row-upt-k m k A*
and *Amk-not-zero: A \$ m \$ from-nat k* $\neq$ *0*
shows $\exists$ *m.* $\neg$ *is-zero-row-upt-k m (Suc k) (Gauss-Jordan-in-ij A 0 (from-nat k))*
proof –
have *to-nat-from-nat-k-suc: to-nat (from-nat k::'columns) < (Suc k)* **using** *to-nat-from-nat-id[OF k[unfolded ncols-def]]* **by** *simp*
have *A0k-eq-1: (Gauss-Jordan-in-ij A 0 (from-nat k)) \$ 0 \$ (from-nat k) = 1*
by *(rule Gauss-Jordan-in-ij-1, auto intro!: Amk-not-zero least-mod-type)*
have $\neg$ *is-zero-row-upt-k 0 (Suc k) (Gauss-Jordan-in-ij A 0 (from-nat k))*
unfolding *is-zero-row-upt-k-def*
using *A0k-eq-1 to-nat-from-nat-k-suc* **by** *force*
thus *?thesis* **by** *blast*
qed

lemma *foldl-Gauss-condition-3*:
fixes *A::'a::{field} ^'columns::{mod-type} ^'rows::{mod-type}* **and** *k::nat*
assumes *k: k < ncols A*
and *all-zero:* $\forall$ *m. is-zero-row-upt-k m k A*
and *Amk-not-zero: A \$ m \$ from-nat k* $\neq$ *0*
and $\neg$ *is-zero-row-upt-k ma (Suc k) (Gauss-Jordan-in-ij A 0 (from-nat k))*
shows *to-nat (GREATEST' n.* $\neg$ *is-zero-row-upt-k n (Suc k) (Gauss-Jordan-in-ij A 0 (from-nat k))) = 0*
proof *(unfold to-nat-eq-0, rule Greatest'-equality)*
have *to-nat-from-nat-k-suc: to-nat (from-nat k::'columns) < Suc (k)* **using** *to-nat-from-nat-id[OF k[unfolded ncols-def]]* **by** *simp*
have *A0k-eq-1: (Gauss-Jordan-in-ij A 0 (from-nat k)) \$ 0 \$ (from-nat k) = 1*
by *(rule Gauss-Jordan-in-ij-1, auto intro!: Amk-not-zero least-mod-type)*
show $\neg$ *is-zero-row-upt-k 0 (Suc k) (Gauss-Jordan-in-ij A 0 (from-nat k))*
unfolding *is-zero-row-upt-k-def*
using *A0k-eq-1 to-nat-from-nat-k-suc* **by** *force*
fix *y*
assume *not-zero-y:* $\neg$ *is-zero-row-upt-k y (Suc k) (Gauss-Jordan-in-ij A 0 (from-nat k))*
have *y-eq-0: y=0*
proof *(rule ccontr)*
assume *y-not-0: y* $\neq$ *0*
have *is-zero-row-upt-k y (Suc k) (Gauss-Jordan-in-ij A 0 (from-nat k))* **unfolding** *is-zero-row-upt-k-def*

proof (*clarify*)
fix *j*::′*columns* **assume** *j*: *to-nat j* < *Suc k*
show *Gauss-Jordan-in-ij A 0* (*from-nat k*) $ *y* $ *j* = *0*
proof (*cases to-nat j* = *k*)
case *True* **show** *?thesis* **unfolding** *to-nat-from-nat*[*OF True*]
by (*rule Gauss-Jordan-in-ij-0*[*OF - y-not-0*], *unfold to-nat-from-nat*[*OF True*, *symmetric*], *auto intro*!: *y-not-0 least-mod-type Amk-not-zero*)
next
case *False* **hence** *j-less-k*: *to-nat j* < *k* **by** (*metis j less-SucE*)
show *?thesis* **using** *Gauss-Jordan-in-ij-preserves-previous-elements*′[*OF all-zero j-less-k Amk-not-zero*]
using *all-zero j-less-k* **unfolding** *is-zero-row-upt-k-def* **by** *presburger*
qed
qed
thus *False* **using** *not-zero-y* **by** *contradiction*
qed
thus *y*≤ *0* **using** *least-mod-type* **by** *simp*
qed

lemma *foldl-Gauss-condition-5*:
fixes *A*::′*a*::{*field*} ^′*columns*::{*mod-type*} ^′*rows*::{*mod-type*} **and** *k*::*nat*
assumes *rref-A*: *reduced-row-echelon-form-upt-k A k*
and *not-zero-a*:¬ *is-zero-row-upt-k a k A*
and *all-zero-below-greatest*: ∀ *m*≥(*GREATEST*′ *n*. ¬ *is-zero-row-upt-k n k A*) + *1*. *A* $ *m* $ *from-nat k* = *0*
shows (*GREATEST*′ *n*. ¬ *is-zero-row-upt-k n k A*) = (*GREATEST*′ *n*. ¬ *is-zero-row-upt-k n* (*Suc k*) *A*)
proof −
have ⋀*n*. (*is-zero-row-upt-k n* (*Suc k*) *A*) = (*is-zero-row-upt-k n k A*)
proof
fix *n* **assume** *is-zero-row-upt-k n* (*Suc k*) *A*
thus *is-zero-row-upt-k n k A* **using** *is-zero-row-upt-k-le* **by** *fast*
next
fix *n* **assume** *zero-n-k*: *is-zero-row-upt-k n k A*
have *n*>(*GREATEST*′ *n*. ¬ *is-zero-row-upt-k n k A*) **by** (*rule greatest-less-zero-row*[*OF rref-A zero-n-k*], *auto intro*!: *not-zero-a*)
hence *n-ge-gratest*: *n* ≥ (*GREATEST*′ *n*. ¬ *is-zero-row-upt-k n k A*) + *1* **using** *le-Suc* **by** *blast*
hence *A-nk-zero*: *A* $ *n* $ (*from-nat k*) = *0* **using** *all-zero-below-greatest* **by** *fast*
show *is-zero-row-upt-k n* (*Suc k*) *A* **by** (*rule is-zero-row-upt-k-suc*[*OF zero-n-k A-nk-zero*])
qed
thus (*GREATEST*′ *n*. ¬ *is-zero-row-upt-k n k A*) = (*GREATEST*′ *n*. ¬ *is-zero-row-upt-k n* (*Suc k*) *A*) **by** *simp*
qed

lemma *foldl-Gauss-condition-6*:
fixes $A::'a::\{field\}\ \hat{}'columns::\{mod\text{-}type\}\ \hat{}'rows::\{mod\text{-}type\}$ **and** $k::nat$
assumes *not-zero-m*: $\neg$ *is-zero-row-upt-k m k A*
and *eq-card*: *Suc* (*to-nat* (*GREATEST' n.* $\neg$ *is-zero-row-upt-k n k A*)) = *nrows A*
shows *nrows A* = *Suc* (*to-nat* (*GREATEST' n.* $\neg$ *is-zero-row-upt-k n* (*Suc k*) *A*))
proof −
have (*GREATEST' n.* $\neg$ *is-zero-row-upt-k n k A*) + *1* = *0* **using** *greatest-plus-one-eq-0*[*OF eq-card*] **.**
hence *greatest-k-eq-minus-1*: (*GREATEST' n.* $\neg$ *is-zero-row-upt-k n k A*) = −*1* **using** *a-eq-minus-1* **by** *blast*
have (*GREATEST' n.* $\neg$ *is-zero-row-upt-k n* (*Suc k*) *A*) = −*1*
proof (*rule Greatest'-equality*)
show $\neg$ *is-zero-row-upt-k* −*1* (*Suc k*) *A*
using *Greatest'I-ex greatest-k-eq-minus-1 is-zero-row-upt-k-le not-zero-m* **by** *force*
show $\bigwedge y.$ $\neg$ *is-zero-row-upt-k y* (*Suc k*) *A* $\Longrightarrow$ *y* $\leq$ −*1* **using** *Greatest-is-minus-1* **by** *fast*
qed
thus *nrows A* = *Suc* (*to-nat* (*GREATEST' n.* $\neg$ *is-zero-row-upt-k n* (*Suc k*) *A*)) **using** *eq-card greatest-k-eq-minus-1* **by** *fastforce*
qed

lemma *foldl-Gauss-condition-8*:
fixes $A::'a::\{field\}\ \hat{}'columns::\{mod\text{-}type\}\ \hat{}'rows::\{mod\text{-}type\}$ **and** $k::nat$
assumes *k*: *k* < *ncols A*
and *not-zero-m*: $\neg$ *is-zero-row-upt-k m k A*
and *A-ma-k*: *A* \$ *ma* \$ *from-nat k* $\neq$ *0*
and *ma*: (*GREATEST' n.* $\neg$ *is-zero-row-upt-k n k A*) + *1* $\leq$ *ma*
shows $\exists$ *m.* $\neg$ *is-zero-row-upt-k m* (*Suc k*) (*Gauss-Jordan-in-ij A* ((*GREATEST' n.* $\neg$ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*))
proof −
def *Greatest-plus-one*≡((*GREATEST' n.* $\neg$ *is-zero-row-upt-k n k A*) + *1*)
have *to-nat-from-nat-k-suc*: *to-nat* (*from-nat k::'columns*) < (*Suc k*) **using** *to-nat-from-nat-id*[*OF k*[*unfolded ncols-def*]] **by** *simp*
have *Gauss-eq-1*: (*Gauss-Jordan-in-ij A Greatest-plus-one* (*from-nat k*)) \$ *Greatest-plus-one* \$ (*from-nat k*) = *1*
by (*unfold Greatest-plus-one-def*, *rule Gauss-Jordan-in-ij-1*, *auto intro*!: *A-ma-k ma*)
show $\exists$ *m.* $\neg$ *is-zero-row-upt-k m* (*Suc k*) (*Gauss-Jordan-in-ij A* (*Greatest-plus-one*) (*from-nat k*))
by (*rule exI*[*of - Greatest-plus-one*], *unfold is-zero-row-upt-k-def*, *auto*, *rule exI*[*of - from-nat k*], *simp add*: *Gauss-eq-1 to-nat-from-nat-k-suc*)
qed

lemma *foldl-Gauss-condition-9*:

fixes *A*::*'a*::{*field*} ^*'columns*::{*mod-type*} ^*'rows*::{*mod-type*} **and** *k*::*nat*
assumes *k*: *k* < *ncols A*
and *rref-A*: *reduced-row-echelon-form-upt-k A k*
assumes *not-zero-m*: ¬ *is-zero-row-upt-k m k A*
and *suc-greatest-not-card*: *Suc* (*to-nat* (*GREATEST' n*. ¬ *is-zero-row-upt-k n k A*)) ≠ *nrows A*
and *greatest-less-ma*: (*GREATEST' n*. ¬ *is-zero-row-upt-k n k A*) + *1* ≤ *ma*
and *A-ma-k*: *A* $ *ma* $ *from-nat k* ≠ *0*
shows *Suc* (*to-nat* (*GREATEST' n*. ¬ *is-zero-row-upt-k n k A*)) =
to-nat(*GREATEST' n*. ¬ *is-zero-row-upt-k n* (*Suc k*) (*Gauss-Jordan-in-ij A* ((*GREATEST' n*. ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*)))
proof −
def *Greatest-plus-one*==((*GREATEST' n*. ¬ *is-zero-row-upt-k n k A*) + *1*)
have *to-nat-from-nat-k-suc*: *to-nat* (*from-nat k*::*'columns*) < (*Suc k*) **using** *to-nat-from-nat-id*[*OF k*[*unfolded ncols-def*]] **by** *simp*
have *greatest-plus-one-not-zero*: *Greatest-plus-one* ≠ *0*
proof −
have *to-nat* (*GREATEST' n*. ¬ *is-zero-row-upt-k n k A*) < *nrows A* **using** *to-nat-less-card* **unfolding** *nrows-def* **by** *blast*
hence *to-nat* (*GREATEST' n*. ¬ *is-zero-row-upt-k n k A*) + *1* < *nrows A* **using** *suc-greatest-not-card* **by** *linarith*
show *?thesis* **unfolding** *Greatest-plus-one-def* **by** (*rule suc-not-zero*[*OF suc-greatest-not-card*[*unfolded Suc-eq-plus1 nrows-def*]])
qed
have *greatest-eq*: *Greatest-plus-one* = (*GREATEST' n*. ¬ *is-zero-row-upt-k n* (*Suc k*) (*Gauss-Jordan-in-ij A Greatest-plus-one* (*from-nat k*)))
proof (*rule Greatest'-equality*[*symmetric*])
have (*Gauss-Jordan-in-ij A Greatest-plus-one* (*from-nat k*)) $ (*Greatest-plus-one*) $ (*from-nat k*) = *1*
by (*unfold Greatest-plus-one-def*, *rule Gauss-Jordan-in-ij-1*, *auto intro*!: *greatest-less-ma A-ma-k*)
thus ¬ *is-zero-row-upt-k Greatest-plus-one* (*Suc k*) (*Gauss-Jordan-in-ij A Greatest-plus-one* (*from-nat k*))
using *to-nat-from-nat-k-suc*
unfolding *is-zero-row-upt-k-def* **by** *fastforce*
fix *y*
assume *not-zero-y*: ¬ *is-zero-row-upt-k y* (*Suc k*) (*Gauss-Jordan-in-ij A Greatest-plus-one* (*from-nat k*))
show *y* ≤ *Greatest-plus-one*
proof (*cases y*<*Greatest-plus-one*)
case *True* **thus** *?thesis* **by** *simp*
next
case *False* **hence** *y-ge-greatest*: *y*≥*Greatest-plus-one* **by** *simp*
have *y*=*Greatest-plus-one*
proof (*rule ccontr*)
assume *y-not-greatest*: *y* ≠ *Greatest-plus-one*
have (*GREATEST' n*. ¬ *is-zero-row-upt-k n k A*) < *y* **using** *greatest-plus-one-not-zero*

using *Suc-le' less-le-trans y-ge-greatest* **unfolding** *Greatest-plus-one-def*

by *auto*
hence *zero-row-y-upt-k*: *is-zero-row-upt-k y k A* **using** *not-greater-Greatest'*[*of* λ*n*. ¬ *is-zero-row-upt-k n k A y*] **unfolding** *Greatest-plus-one-def* **by** *fast*
have *is-zero-row-upt-k y* (*Suc k*) (*Gauss-Jordan-in-ij A Greatest-plus-one* (*from-nat k*)) **unfolding** *is-zero-row-upt-k-def*
proof (*clarify*)
fix *j*::*'columns* **assume** *j*: *to-nat j* < *Suc k*
show *Gauss-Jordan-in-ij A Greatest-plus-one* (*from-nat k*) $ *y* $ *j* = *0*
proof (*cases j*=*from-nat k*)
case *True*
show *?thesis*
proof (*unfold True*, *rule Gauss-Jordan-in-ij-0*[*OF* - *y-not-greatest*], *rule exI*[*of* - *ma*], *rule conjI*)
show *A* $ *ma* $ *from-nat k* ≠ *0* **using** *A-ma-k* **.**
show *Greatest-plus-one* ≤ *ma* **using** *greatest-less-ma* **unfolding** *Greatest-plus-one-def* **.**
qed
next
case *False* **hence** *j-le-suc-k*: *to-nat j* < *Suc k* **using** *j* **by** *simp*
have *Gauss-Jordan-in-ij A Greatest-plus-one* (*from-nat k*) $ *y* $ *j* = *A* $ *y* $ *j* **unfolding** *Greatest-plus-one-def*
proof (*rule Gauss-Jordan-in-ij-preserves-previous-elements*)
show *reduced-row-echelon-form-upt-k A k* **using** *rref-A* **.**
show ¬ *is-zero-row-upt-k m k A* **using** *not-zero-m* **.**
show ∃ *n*. *A* $ *n* $ *from-nat k* ≠ *0* ∧ (*GREATEST' n*. ¬ *is-zero-row-upt-k n k A*) + *1* ≤ *n* **using** *A-ma-k greatest-less-ma* **by** *blast*
show (*GREATEST' n*. ¬ *is-zero-row-upt-k n k A*) + *1* ≠ *0* **using** *greatest-plus-one-not-zero* **unfolding** *Greatest-plus-one-def* **.**
show *to-nat j* < *k* **using** *False from-nat-to-nat-id j-le-suc-k less-antisym* **by** *fastforce*
qed
also have ... = *0* **using** *zero-row-y-upt-k* **unfolding** *is-zero-row-upt-k-def* **using** *False le-imp-less-or-eq from-nat-to-nat-id j-le-suc-k less-Suc-eq-le* **by** *fastforce*
finally show *Gauss-Jordan-in-ij A Greatest-plus-one* (*from-nat k*) $ *y* $ *j* = *0* **.**
qed
qed
thus *False* **using** *not-zero-y* **by** *contradiction*
qed
thus *y* ≤ *Greatest-plus-one* **using** *y-ge-greatest* **by** *blast*
qed
qed
show *Suc* (*to-nat* (*GREATEST' n*. ¬ *is-zero-row-upt-k n k A*)) =
to-nat (*GREATEST' n*. ¬ *is-zero-row-upt-k n* (*Suc k*) (*Gauss-Jordan-in-ij A* ((*GREATEST' n*. ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*)))
unfolding *greatest-eq*[*unfolded Greatest-plus-one-def*, *symmetric*]
unfolding *add-to-nat-def*
unfolding *to-nat-1*

using *to-nat-from-nat-id to-nat-plus-one-less-card*
using *greatest-plus-one-not-zero*[*unfolded Greatest-plus-one-def*]
by *force*
qed

The following lemma is one of most important ones in the verification of the Gauss-Jordan algorithm. The aim is to prove two statements about *Gauss-Jordan-upt-k ?A ?k* = *snd* (*foldl Gauss-Jordan-column-k* (*0*, *?A*) [*0*..<*Suc ?k*]) (one about the result is on rref and another about the index). The reason of doing that way is because both statements need them mutually to be proved. As the proof is made using induction, two base casis and two induction steps appear.

lemma *rref-and-index-Gauss-Jordan-upt-k*:
fixes *A*::′*a*::{*field*} ^′*columns*::{*mod-type*} ^′*rows*::{*mod-type*} **and** *k*::*nat*
assumes *k* < *ncols A*
shows *rref-Gauss-Jordan-upt-k*: *reduced-row-echelon-form-upt-k* (*Gauss-Jordan-upt-k A k*) (*Suc k*)
and *snd-Gauss-Jordan-upt-k*:
foldl Gauss-Jordan-column-k (*0*, *A*) [*0*..<*Suc k*] =
(*if* ∀ *m*. *is-zero-row-upt-k m* (*Suc k*) (*snd* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc k*])) *then 0*
else to-nat (*GREATEST*′ *n*. ¬ *is-zero-row-upt-k n* (*Suc k*) (*snd* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc k*]))) + *1*,
snd (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc k*]))
using *assms*
proof (*induct k*)
— Two base cases, one for each show
— The first one
show *reduced-row-echelon-form-upt-k* (*Gauss-Jordan-upt-k A 0*) (*Suc 0*)
unfolding *Gauss-Jordan-upt-k-def* **apply** *auto*
using *reduced-row-echelon-form-upt-k-Gauss-Jordan-column-k*[*OF rref-upt-0*, *of A*] **using** *is-zero-row-utp-0*′[*of A*] **by** *simp*
— The second base case
have *rw-upt*: [*0*..<*Suc 0*] = [*0*] **by** *simp*
show *foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc 0*] =
(*if* ∀ *m*. *is-zero-row-upt-k m* (*Suc 0*) (*snd* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc 0*])) *then 0*
else to-nat (*GREATEST*′ *n*. ¬ *is-zero-row-upt-k n* (*Suc 0*) (*snd* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc 0*]))) + *1*,
snd (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc 0*]))
unfolding *rw-upt*
unfolding *foldl.simps*
unfolding *Gauss-Jordan-column-k-def Let-def from-nat-0 fst-conv snd-conv*
unfolding *is-zero-row-upt-k-def*
apply (*auto simp add*: *least-mod-type to-nat-eq-0*)
apply (*metis Gauss-Jordan-in-ij-1 least-mod-type zero-neq-one*)
by (*metis* (*lifting*, *mono-tags*) *Gauss-Jordan-in-ij-0 Greatest*′*I-ex least-mod-type*)
next

— Now we begin with the proof of the induction step of the first show. We will make use the induction hypothesis of the second show

fix *k*

assume ($k < ncols\ A \Longrightarrow$ *reduced-row-echelon-form-upt-k* (*Gauss-Jordan-upt-k* *A k*) (*Suc k*))

and ($k < ncols\ A \Longrightarrow$

foldl Gauss-Jordan-column-k (*0*, *A*) [*0*..<*Suc k*] =

(*if* ∀ *m*. *is-zero-row-upt-k m* (*Suc k*) (*snd* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc k*])) *then 0*

else to-nat (*GREATEST'* *n*. ¬ *is-zero-row-upt-k n* (*Suc k*) (*snd* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc k*]))) + *1*,

snd (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc k*])))

and *k*: $Suc\ k < ncols\ A$

hence *hyp-rref*: *reduced-row-echelon-form-upt-k* (*Gauss-Jordan-upt-k A k*) (*Suc k*)

and *hyp-foldl*: *foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc k*] =

(*if* ∀ *m*. *is-zero-row-upt-k m* (*Suc k*) (*snd* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc k*])) *then 0*

else to-nat (*GREATEST'* *n*. ¬ *is-zero-row-upt-k n* (*Suc k*) (*snd* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc k*]))) + *1*,

snd (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc k*]))

by *simp+*

have *rw*: [*0*..<*Suc* (*Suc k*)]= [*0*..<(*Suc k*)] @ [(*Suc k*)] **by** *auto*

have *rw2*: (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc k*]) =

(*if* ∀ *m*. *is-zero-row-upt-k m* (*Suc k*) (*Gauss-Jordan-upt-k A k*) *then 0 else to-nat* (*GREATEST'* *n*. ¬ *is-zero-row-upt-k n* (*Suc k*) (*Gauss-Jordan-upt-k A k*)) + *1*,

Gauss-Jordan-upt-k A k) **unfolding** *Gauss-Jordan-upt-k-def* **using** *hyp-foldl* **by** *fast*

show *reduced-row-echelon-form-upt-k* (*Gauss-Jordan-upt-k A* (*Suc k*)) (*Suc* (*Suc k*))

unfolding *Gauss-Jordan-upt-k-def* **unfolding** *rw* **unfolding** *foldl-append* **unfolding** *foldl.simps* **unfolding** *rw2*

by (*rule reduced-row-echelon-form-upt-k-Gauss-Jordan-column-k*[*OF hyp-rref*])

— Making use of the same hypotheses of above proof, we begin with the proof of the induction step of the second show.

have *fst-foldl*: *fst* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc k*]) =

fst (*if* ∀ *m*. *is-zero-row-upt-k m* (*Suc k*) (*snd* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc k*])) *then 0*

else to-nat (*GREATEST'* *n*. ¬ *is-zero-row-upt-k n* (*Suc k*) (*snd* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc k*]))) + *1*,

snd (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc k*])) **using** *hyp-foldl* **by** *simp*

show *foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc* (*Suc k*)] =

(*if* ∀ *m*. *is-zero-row-upt-k m* (*Suc* (*Suc k*)) (*snd* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc* (*Suc k*)])) *then 0*

else to-nat (*GREATEST'* *n*. ¬ *is-zero-row-upt-k n* (*Suc* (*Suc k*)) (*snd* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc* (*Suc k*)]))) + *1*,

snd (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc* (*Suc k*)]))

proof (*rule prod-eqI*)

show *snd* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc* (*Suc k*)]) =

snd (*if* ∀ *m*. *is-zero-row-upt-k m* (*Suc* (*Suc k*)) (*snd* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc* (*Suc k*)])) *then 0*
else to-nat (*GREATEST′ n*. ¬ *is-zero-row-upt-k n* (*Suc* (*Suc k*)) (*snd* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc* (*Suc k*)]))) + *1*,
snd (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc* (*Suc k*)]))
unfolding *Gauss-Jordan-upt-k-def* **by** *force*
def *A′*≡(*snd* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc k*]))
have *ncols-eq*: *ncols A* = *ncols A′* **unfolding** *A′-def ncols-def* **..**
have *rref-A′*: *reduced-row-echelon-form-upt-k A′* (*Suc k*) **using** *hyp-rref* **unfolding** *A′-def Gauss-Jordan-upt-k-def* **.**
show *fst* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc* (*Suc k*)]) =
fst (*if* ∀ *m*. *is-zero-row-upt-k m* (*Suc* (*Suc k*)) (*snd* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc* (*Suc k*)])) *then 0*
else to-nat (*GREATEST′ n*. ¬ *is-zero-row-upt-k n* (*Suc* (*Suc k*)) (*snd* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc* (*Suc k*)]))) + *1*,
snd (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc* (*Suc k*)]))
unfolding *rw* **unfolding** *foldl-append* **unfolding** *foldl.simps* **unfolding** *Gauss-Jordan-column-k-def Let-def fst-foldl* **unfolding** *A′-def*[*symmetric*]
proof (*auto*, *unfold from-nat-0 from-nat-to-nat-greatest*)
fix *m* **assume** ∀ *m*. *is-zero-row-upt-k m* (*Suc k*) *A′* **and** ∀ *m*≥*0*. *A′* $ *m* $ *from-nat* (*Suc k*) = *0*
thus *is-zero-row-upt-k m* (*Suc* (*Suc k*)) *A′* **using** *foldl-Gauss-condition-1* **by** *blast*
next
fix *m*
assume ∀ *m*. *is-zero-row-upt-k m* (*Suc k*) *A′*
and *A′* $ *m* $ *from-nat* (*Suc k*) ≠ *0*
thus ∃ *m*. ¬ *is-zero-row-upt-k m* (*Suc* (*Suc k*)) (*Gauss-Jordan-in-ij A′ 0* (*from-nat* (*Suc k*)))
using *foldl-Gauss-condition-2 k ncols-eq* **by** *simp*
next
fix *m ma*
assume ∀ *m*. *is-zero-row-upt-k m* (*Suc k*) *A′*
and *A′* $ *m* $ *from-nat* (*Suc k*) ≠ *0*
and ¬ *is-zero-row-upt-k ma* (*Suc* (*Suc k*)) (*Gauss-Jordan-in-ij A′ 0* (*from-nat* (*Suc k*)))
thus *to-nat* (*GREATEST′ n*. ¬ *is-zero-row-upt-k n* (*Suc* (*Suc k*)) (*Gauss-Jordan-in-ij A′ 0* (*from-nat* (*Suc k*)))) = *0*
using *foldl-Gauss-condition-3 k ncols-eq* **by** *simp*
next
fix *m* **assume** ¬ *is-zero-row-upt-k m* (*Suc k*) *A′*
thus ∃ *m*. ¬ *is-zero-row-upt-k m* (*Suc* (*Suc k*)) *A′* **and** ∃ *m*. ¬ *is-zero-row-upt-k m* (*Suc* (*Suc k*)) *A′* **using** *is-zero-row-upt-k-le* **by** *blast+*
next
fix *m*
assume *not-zero-m*: ¬ *is-zero-row-upt-k m* (*Suc k*) *A′*
and *zero-below-greatest*: ∀ *m*≥(*GREATEST′ n*. ¬ *is-zero-row-upt-k n* (*Suc k*) *A′*) + *1*. *A′* $ *m* $ *from-nat* (*Suc k*) = *0*
show (*GREATEST′ n*. ¬ *is-zero-row-upt-k n* (*Suc k*) *A′*) = (*GREATEST′*

n. ¬ *is-zero-row-upt-k n* (*Suc* (*Suc k*)) *A′*)
  by (*rule foldl-Gauss-condition-5*[*OF rref-A′ not-zero-m zero-below-greatest*])
 next
 fix *m* **assume** ¬ *is-zero-row-upt-k m* (*Suc k*) *A′* **and** *Suc* (*to-nat* (*GREATEST′ n*. ¬ *is-zero-row-upt-k n* (*Suc k*) *A′*)) = *nrows A′*
  thus *nrows A′* = *Suc* (*to-nat* (*GREATEST′ n*. ¬ *is-zero-row-upt-k n* (*Suc* (*Suc k*)) *A′*))
   using *foldl-Gauss-condition-6* **by** *blast*
 next
  fix *m ma*
  assume ¬ *is-zero-row-upt-k m* (*Suc k*) *A′*
   and (*GREATEST′ n*. ¬ *is-zero-row-upt-k n* (*Suc k*) *A′*) + *1* ≤ *ma*
   and *A′* $ *ma* $ *from-nat* (*Suc k*) ≠ *0*
   thus ∃ *m*. ¬ *is-zero-row-upt-k m* (*Suc* (*Suc k*)) (*Gauss-Jordan-in-ij A′* ((*GREATEST′ n*. ¬ *is-zero-row-upt-k n* (*Suc k*) *A′*) + *1*) (*from-nat* (*Suc k*)))
   using *foldl-Gauss-condition-8* **using** *k ncols-eq* **by** *simp*
 next
  fix *m ma mb*
  assume ¬ *is-zero-row-upt-k m* (*Suc k*) *A′* **and**
   Suc (*to-nat* (*GREATEST′ n*. ¬ *is-zero-row-upt-k n* (*Suc k*) *A′*)) ≠ *nrows A′*
   and (*GREATEST′ n*. ¬ *is-zero-row-upt-k n* (*Suc k*) *A′*) + *1* ≤ *ma*
   and *A′* $ *ma* $ *from-nat* (*Suc k*) ≠ *0*
  and ¬ *is-zero-row-upt-k mb* (*Suc* (*Suc k*)) (*Gauss-Jordan-in-ij A′* ((*GREATEST′ n*. ¬ *is-zero-row-upt-k n* (*Suc k*) *A′*) + *1*) (*from-nat* (*Suc k*)))
  thus *Suc* (*to-nat* (*GREATEST′ n*. ¬ *is-zero-row-upt-k n* (*Suc k*) *A′*)) =
  to-nat (*GREATEST′ n*. ¬ *is-zero-row-upt-k n* (*Suc* (*Suc k*)) (*Gauss-Jordan-in-ij A′* ((*GREATEST′ n*. ¬ *is-zero-row-upt-k n* (*Suc k*) *A′*) + *1*) (*from-nat* (*Suc k*))))
   using *foldl-Gauss-condition-9*[*OF k*[*unfolded ncols-eq*] *rref-A′*] **unfolding** *nrows-def* **by** *blast*
 qed
 qed
qed

corollary *rref-Gauss-Jordan*:
 fixes *A*::′*a*::{*field*} ^′*columns*::{*mod-type*} ^′*rows*::{*mod-type*}
 shows *reduced-row-echelon-form* (*Gauss-Jordan A*)
proof −
 have *CARD*(′*columns*) − *1* < *CARD*(′*columns*) **by** *fastforce*
 thus *reduced-row-echelon-form* (*Gauss-Jordan A*)
  unfolding *reduced-row-echelon-form-def Gauss-Jordan-def*
  using *rref-Gauss-Jordan-upt-k* **unfolding** *ncols-def* **by** *fastforce*
qed

lemma *independent-not-zero-rows-rref*:
 fixes *A*::*real* ^′*m*::{*mod-type*} ^′*n*::{*finite*,*one*,*plus*,*ord*}
 assumes *rref-A*: *reduced-row-echelon-form A*

shows *independent* $\{$*row i A* $|$*i. row i A* $\neq$ *0*$\}$
proof
def $R \equiv \{$*row i A* $|$*i. row i A* $\neq$ *0*$\}$
assume *dep*: *dependent R*
from *this* **obtain** *a* **where** *a-in-R*: *a*$\in$*R* **and** *a-in-span*: *a* $\in$ *span* (*R* $-$ $\{a\}$) **unfolding** *dependent-def* **by** *fast*
from *a-in-R* **obtain** *i* **where** *a-eq-row-i-A*: *a=row i A* **unfolding** *R-def* **by** *blast*
hence *a-eq-Ai*: *a* = *A* \$ *i* **unfolding** *row-def* **unfolding** *vec-nth-inverse* **.**
have *row-i-A-not-zero*: $\neg$ *is-zero-row i A* **using** *a-in-R*
unfolding *R-def is-zero-row-def is-zero-row-upt-ncols row-def vec-nth-inverse*
unfolding *vec-lambda-unique zero-vec-def mem-Collect-eq* **using** *a-eq-Ai* **by** *force*
def *least-n* == (*LEAST n. A* \$ *i* \$ *n* $\neq$ *0*)
have *span-rw*: *span* (*R* $-$ $\{a\}$) = $\{$*y.* $\exists$ *u.* ($\sum$ *v*$\in$(*R* $-$ $\{a\}$). *u v* $*_R$ *v*) = *y*$\}$
proof (*rule span-finite*)
show *finite* (*R* $-$ $\{a\}$) **using** *finite-rows*[*of A*] **unfolding** *rows-def R-def* **by** *simp*
qed
from *this* **obtain** *f* **where** *f*: ($\sum$ *v*$\in$(*R* $-$ $\{a\}$). *f v* $*_R$ *v*) = *a* **using** *a-in-span* **by** *fast*
have *1* = *a* \$ *least-n* **using** *rref-condition2*[*OF rref-A*] *row-i-A-not-zero* **unfolding** *least-n-def a-eq-Ai* **by** *presburger*
also have... = ($\sum$ *v*$\in$(*R* $-$ $\{a\}$). *f v* $*_R$ *v*) \$ *least-n* **using** *f* **by** *auto*
also have ... = ($\sum$ *v*$\in$(*R* $-$ $\{a\}$). (*f v* $*_R$ *v*) \$ *least-n*) **unfolding** *setsum-component* **..**
also have ... = ($\sum$ *v*$\in$(*R* $-$ $\{a\}$). *f v* $*_R$ (*v* \$ *least-n*)) **unfolding** *vector-scaleR-component* **..**
also have ... = ($\sum$ *v*$\in$(*R* $-$ $\{a\}$). *0*)
proof (*rule setsum-cong2*)
fix *x* **assume** *x*: *x* $\in$ *R* $-$ $\{a\}$
from *this* **obtain** *j* **where** *x-eq-row-j-A*: *x=row j A* **unfolding** *R-def* **by** *auto*
hence *i-not-j*: *i* $\neq$ *j* **by** (*metis a-eq-row-i-A mem-delete x*)
have *x-least-is-zero*: *x* \$ *least-n* = *0* **using** *rref-condition4*[*OF rref-A*] *i-not-j row-i-A-not-zero*
unfolding *x-eq-row-j-A least-n-def row-def vec-nth-inverse* **by** *blast*
show *f x* $*_R$ *x* \$ *least-n* = *0* **unfolding** *x-least-is-zero scaleR-zero-right* **..**
qed
also have ... = *0* **unfolding** *setsum-0* **..**
finally show *False* **by** *simp*
qed

lemma *rref-rank*:
fixes *A*::*real*^'*m*::$\{$*mod-type*$\}$^'*n*::$\{$*finite,one,plus,ord*$\}$
assumes *rref-A*: *reduced-row-echelon-form A*
shows *rank A* = *card* $\{$*row i A* $|$*i. row i A* $\neq$ *0*$\}$
unfolding *rank-def row-rank-def*
proof (*rule dim-unique*[*of* $\{$*row i A* $|$ *i. row i A* $\neq$ *0*$\}$])
show $\{$*row i A* $|$*i. row i A* $\neq$ *0*$\}$ $\subseteq$ *row-space A*
proof (*auto*, *unfold row-space-def rows-def*)
fix *i* **assume** *row i A* $\neq$ *0* **show** *row i A* $\in$ *span* $\{$*row i A* $|$*i. i* $\in$ *UNIV*$\}$ **by**

(*rule span-superset*, *auto*)
  qed
  show *row-space A ⊆ span {row i A |i. row i A ≠ 0}*
  proof (*unfold row-space-def rows-def*, *cases ∃ i. row i A = 0*)
    case *True*
     have *set-rw*: {*row i A* |*i. i ∈ UNIV*} = *insert 0* {*row i A* |*i. row i A ≠ 0*}
using *True* **by** *auto*
    have *span* {*row i A* |*i. i ∈ UNIV*} = *span* {*row i A* |*i. row i A ≠ 0*} **unfolding**
set-rw **using** *span-insert-0* **.**
    thus *span* {*row i A* |*i. i ∈ UNIV*} ⊆ *span* {*row i A* |*i. row i A ≠ 0*} **by** *simp*
  next
    case *False* **show** *span* {*row i A* |*i. i ∈ UNIV*} ⊆ *span* {*row i A* |*i. row i A ≠
0*} **using** *False* **by** *simp*
  qed
 show *independent* {*row i A* |*i. row i A ≠ 0*} **by** (*rule independent-not-zero-rows-rref*[*OF
rref-A*])
  show *card* {*row i A* |*i. row i A ≠ 0*} = *card* {*row i A* |*i. row i A ≠ 0*} **..**
qed

This function eliminates all entries of the j-th column using the non-zero element situated in the position (i,j). It is introduced to make easier the proof that each Gauss-Jordan step consists in applying suitable elementary operations.

primrec *row-add-iterate* :: *'a::{semiring-1, uminus}^'n^'m::{mod-type} => nat
=> 'm => 'n => 'a^'n^'m::{mod-type}*
  where *row-add-iterate A 0 i j* = (*if i=0 then A else row-add A 0 i* (−*A* $ *0* $
j))
  | *row-add-iterate A* (*Suc n*) *i j* = (*if* (*Suc n* = *to-nat i*) *then row-add-iterate A
n i j*
      else row-add-iterate (*row-add A* (*from-nat* (*Suc n*)) *i* (− *A* $ (*from-nat* (*Suc
n*)) $ *j*)) *n i j*)

lemma *invertible-row-add-iterate*:
  fixes *A::'a::{ring-1}^'n^'m::{mod-type}*
  assumes *n*: *n<nrows A*
  shows *∃ P. invertible P ∧ row-add-iterate A n i j = P**A*
  using *n*
proof (*induct n arbitrary*: *A*)
  fix *A::'a::{ring-1}^'n^'m::{mod-type}*
  show *∃ P. invertible P ∧ row-add-iterate A 0 i j = P ** A*
  proof (*cases i=0*)
    case *True* **show** *?thesis*
     unfolding *row-add-iterate.simps* **by** (*metis True invertible-def matrix-mul-lid*)
  next
    case *False*
   show *?thesis* **by** (*metis False invertible-row-add row-add-iterate.simps*(*1*) *row-add-mat-1*)

  qed
  fix *n* **and** *A::'a::{ring-1}^'n^'m::{mod-type}*

def *A'*==(*row-add A* (*from-nat* (*Suc n*)) *i* (− *A* \$ *from-nat* (*Suc n*) \$ *j*))
assume *hyp*: ⋀*A*::'*a*::{*ring-1*} ^'*n* ^'*m*::{*mod-type*}. *n* < *nrows A* ⟹ ∃ *P*. *invertible P* ∧ *row-add-iterate A n i j* = *P* ** *A* **and** *Suc-n*: *Suc n* < *nrows A*
hence ∃ *P*. *invertible P* ∧ *row-add-iterate A' n i j* = *P* ** *A'* **unfolding** *nrows-def* **by** *auto*
from *this* **obtain** *P* **where** *inv-P*: *invertible P* **and** *P*: *row-add-iterate A' n i j* = *P* ** *A'* **by** *auto*
show ∃ *P*. *invertible P* ∧ *row-add-iterate A* (*Suc n*) *i j* = *P* ** *A*
unfolding *row-add-iterate.simps*
proof (*cases Suc n* = *to-nat i*)
case *True*
show ∃ *P*. *invertible P* ∧
(*if Suc n* = *to-nat i then row-add-iterate A n i j*
else row-add-iterate (*row-add A* (*from-nat* (*Suc n*)) *i* (− *A* \$ *from-nat* (*Suc n*) \$ *j*)) *n i j*) =
P ** *A*
unfolding *if-P*[*OF True*] **using** *hyp Suc-n* **by** *simp*
next
case *False*
show ∃ *P*. *invertible P* ∧
(*if Suc n* = *to-nat i then row-add-iterate A n i j*
else row-add-iterate (*row-add A* (*from-nat* (*Suc n*)) *i* (− *A* \$ *from-nat* (*Suc n*) \$ *j*)) *n i j*) =
P ** *A*
unfolding *if-not-P*[*OF False*]
unfolding *P*[*unfolded A'-def*]
proof (*rule exI*[*of - P* ** (*row-add* (*mat 1*) (*from-nat* (*Suc n*)) *i* (− *A* \$ *from-nat* (*Suc n*) \$ *j*))], *rule conjI*)
show *invertible* (*P* ** *row-add* (*mat 1*) (*from-nat* (*Suc n*)) *i* (− *A* \$ *from-nat* (*Suc n*) \$ *j*))
by (*metis False Suc-n inv-P invertible-mult invertible-row-add to-nat-from-nat-id nrows-def*)
show *P* ** *row-add A* (*from-nat* (*Suc n*)) *i* (− *A* \$ *from-nat* (*Suc n*) \$ *j*) =
P ** *row-add* (*mat 1*) (*from-nat* (*Suc n*)) *i* (− *A* \$ *from-nat* (*Suc n*) \$ *j*) ** *A*
using *matrix-mul-assoc row-add-mat-1*[*of from-nat* (*Suc n*) *i* (− *A* \$ *from-nat* (*Suc n*) \$ *j*)]
by *metis*
qed
qed
qed

lemma *row-add-iterate-preserves-greater-than-n*:
fixes *A*::'*a*::{*ring-1*} ^'*n* ^'*m*::{*mod-type*}
assumes *n*: *n*<*nrows A*
and *a*: *to-nat a* > *n*
shows (*row-add-iterate A n i j*) \$ *a* \$ *b* = *A* \$ *a* \$ *b*
using *assms*
proof (*induct n arbitrary*: *A*)

case *0*
show *?case* **unfolding** *row-add-iterate.simps*
proof (*auto*)
assume $i \neq 0$
hence $a \neq 0$ **by** (*metis 0.prems(2) less-numeral-extra(3) to-nat-0*)
thus *row-add A 0 i* (− *A* $ *0* $ *j*) $ *a* $ *b* = *A* $ *a* $ *b* **unfolding** *row-add-def* **by** *auto*
qed
next
fix *n* **and** *A*::′*a*::{*ring-1*} ^′*n* ^′*m*::{*mod-type*}
assume *hyp*: (⋀*A*::′*a*::{*ring-1*} ^′*n* ^′*m*::{*mod-type*}. *n* < *nrows A* ⟹ *n* < *to-nat a* ⟹ *row-add-iterate A n i j* $ *a* $ *b* = *A* $ *a* $ *b*)
and *suc-n-less-card*: *Suc n* < *nrows A* **and** *suc-n-kess-a*: *Suc n* < *to-nat a*
hence *row-add-iterate-A*: *row-add-iterate A n i j* $ *a* $ *b* = *A* $ *a* $ *b* **by** *auto*
show *row-add-iterate A* (*Suc n*) *i j* $ *a* $ *b* = *A* $ *a* $ *b*
proof (*cases Suc n* = *to-nat i*)
case *True*
show *row-add-iterate A* (*Suc n*) *i j* $ *a* $ *b* = *A* $ *a* $ *b* **unfolding** *row-add-iterate.simps if-P*[*OF True*] **using** *row-add-iterate-A* **.**
next
case *False*
def *A*′ ≡ *row-add A* (*from-nat* (*Suc n*)) *i* (− *A* $ *from-nat* (*Suc n*) $ *j*)
have *row-add-iterate-A*′: *row-add-iterate A*′ *n i j* $ *a* $ *b* = *A*′ $ *a* $ *b* **using** *hyp suc-n-less-card suc-n-kess-a* **unfolding** *nrows-def* **by** *auto*
have *from-nat-not-a*: *from-nat* (*Suc n*) ≠ *a* **by** (*metis less-not-refl suc-n-kess-a suc-n-less-card to-nat-from-nat-id nrows-def*)
show *row-add-iterate A* (*Suc n*) *i j* $ *a* $ *b* = *A* $ *a* $ *b* **unfolding** *row-add-iterate.simps if-not-P*[*OF False*] *row-add-iterate-A*′[*unfolded A*′*-def*]
unfolding *row-add-def* **using** *from-nat-not-a* **by** *simp*
qed
qed

lemma *row-add-iterate-preserves-pivot-row*:
fixes *A*::′*a*::{*ring-1*} ^′*n* ^′*m*::{*mod-type*}
assumes *n*: *n*<*nrows A*
and *a*: *to-nat i* ≤ *n*
shows (*row-add-iterate A n i j*) $ *i* $ *b* = *A* $ *i* $ *b*
using *assms*
proof (*induct n arbitrary*: *A*)
case *0*
show *?case* **by** (*metis 0.prems(2) le-0-eq least-mod-type row-add-iterate.simps(1) to-nat-eq to-nat-mono*′)
next
fix *n* **and** *A*::′*a*::{*ring-1*} ^′*n* ^′*m*::{*mod-type*}
assume *hyp*: ⋀*A*::′*a*::{*ring-1*} ^′*n* ^′*m*::{*mod-type*}. *n* < *nrows A* ⟹ *to-nat i* ≤ *n* ⟹ *row-add-iterate A n i j* $ *i* $ *b* = *A* $ *i* $ *b*
and *Suc-n-less-card*: *Suc n* < *nrows A* **and** *i-less-suc*: *to-nat i* ≤ *Suc n*
show *row-add-iterate A* (*Suc n*) *i j* $ *i* $ *b* = *A* $ *i* $ *b*

proof (*cases Suc n* = *to-nat i*)
case *True*
show *?thesis* **unfolding** *row-add-iterate.simps if-P*[*OF True*] **apply** (*rule row-add-iterate-preserves-greater-than-n*) **using** *Suc-n-less-card True lessI* **by** *linarith+*
next
case *False*
def $A' \equiv$ (*row-add A* (*from-nat* (*Suc n*)) *i* ($-$ *A* \$ *from-nat* (*Suc n*) \$ *j*))
have *row-add-iterate-A'*: *row-add-iterate A' n i j* \$ *i* \$ *b* = *A'* \$ *i* \$ *b* **using** *hyp Suc-n-less-card i-less-suc False* **unfolding** *nrows-def* **by** *auto*
have *from-nat-noteq-i*: *from-nat* (*Suc n*) $\neq$ *i* **using** *False Suc-n-less-card from-nat-not-eq* **unfolding** *nrows-def* **by** *blast*
show *?thesis* **unfolding** *row-add-iterate.simps if-not-P*[*OF False*] *row-add-iterate-A'*[*unfolded A'-def*]
unfolding *row-add-def* **using** *from-nat-noteq-i* **by** *simp*
qed
qed

lemma *row-add-iterate-eq-row-add*:
fixes *A*::*'a*::{*ring-1*}^*'n*^*'m*::{*mod-type*}
assumes *a-not-i*: *a* $\neq$ *i*
and *n*: *n*<*nrows A*
and *to-nat a* $\leq$ *n*
shows (*row-add-iterate A n i j*) \$ *a* \$ *b* = (*row-add A a i* ($-$ *A* \$ *a* \$ *j*)) \$ *a* \$ *b*
using *assms*
proof (*induct n arbitrary*: *A*)
case *0*
show *?case* **unfolding** *row-add-iterate.simps* **using** *0.prems*(*3*) *a-not-i to-nat-eq-0 least-mod-type* **by** *force*
next
fix *n* **and** *A*::*'a*::{*ring-1*}^*'n*^*'m*::{*mod-type*}
assume *hyp*: ($\bigwedge$*A*::*'a*::{*ring-1*}^*'n*^*'m*::{*mod-type*}. *a* $\neq$ *i* $\Longrightarrow$ *n* < *nrows A* $\Longrightarrow$ *to-nat a* $\leq$ *n*
$\Longrightarrow$ *row-add-iterate A n i j* \$ *a* \$ *b* = *row-add A a i* ($-$ *A* \$ *a* \$ *j*) \$ *a* \$ *b*)
and *a-not-i*: *a* $\neq$ *i*
and *suc-n-less-card*: *Suc n* < *nrows A*
and *a-le-suc-n*: *to-nat a* $\leq$ *Suc n*
show *row-add-iterate A* (*Suc n*) *i j* \$ *a* \$ *b* = *row-add A a i* ($-$ *A* \$ *a* \$ *j*) \$ *a* \$ *b*
proof (*cases Suc n* = *to-nat i*)
case *True*
show *row-add-iterate A* (*Suc n*) *i j* \$ *a* \$ *b* = *row-add A a i* ($-$ *A* \$ *a* \$ *j*) \$ *a* \$ *b* **unfolding** *row-add-iterate.simps if-P*[*OF True*]
apply (*rule hyp*[*OF a-not-i*], *auto simp add*: *Suc-lessD suc-n-less-card*) **by** (*metis True a-le-suc-n a-not-i le-SucE to-nat-eq*)
next
case *False* **note** *Suc-n-not-i=False*
show *?thesis* **unfolding** *row-add-iterate.simps if-not-P*[*OF False*]

proof (*cases to-nat a = Suc n*) **case** *True*
show *row-add-iterate* (*row-add A* (*from-nat* (*Suc n*)) *i* (− *A* \$ *from-nat* (*Suc n*) \$ *j*)) *n i j* \$ *a* \$ *b* = *row-add A a i* (− *A* \$ *a* \$ *j*) \$ *a* \$ *b*
by (*metis Suc-le-lessD True dual-order.order-refl less-imp-le row-add-iterate-preserves-greater-than-n suc-n-less-card to-nat-from-nat nrows-def*)
next
case *False*
def *A′*≡(*row-add A* (*from-nat* (*Suc n*)) *i* (− *A* \$ *from-nat* (*Suc n*) \$ *j*))
have *rw*: *row-add-iterate A′ n i j* \$ *a* \$ *b* = *row-add A′ a i* (− *A′* \$ *a* \$ *j*) \$ *a* \$ *b*
proof (*rule hyp*)
show *a* ≠ *i* **using** *a-not-i* **.**
show *n* < *nrows A′* **using** *suc-n-less-card* **unfolding** *nrows-def* **by** *auto*
show *to-nat a* ≤ *n* **using** *False a-le-suc-n* **by** *simp*
qed
have *rw1*: *row-add A* (*from-nat* (*Suc n*)) *i* (− *A* \$ *from-nat* (*Suc n*) \$ *j*) \$ *a* \$ *b* = *A* \$ *a* \$ *b*
unfolding *row-add-def* **using** *False suc-n-less-card* **unfolding** *nrows-def* **by** (*auto simp add*: *to-nat-from-nat-id*)
have *rw2*: *row-add A* (*from-nat* (*Suc n*)) *i* (− *A* \$ *from-nat* (*Suc n*) \$ *j*) \$ *a* \$ *j* = *A* \$ *a* \$ *j*
unfolding *row-add-def* **using** *False suc-n-less-card* **unfolding** *nrows-def* **by** (*auto simp add*: *to-nat-from-nat-id*)
have *rw3*: *row-add A* (*from-nat* (*Suc n*)) *i* (− *A* \$ *from-nat* (*Suc n*) \$ *j*) \$ *i* \$ *b* = *A* \$ *i* \$ *b*
unfolding *row-add-def* **using** *Suc-n-not-i suc-n-less-card* **unfolding** *nrows-def* **by** (*auto simp add*: *to-nat-from-nat-id*)
show *row-add-iterate A′ n i j* \$ *a* \$ *b* = *row-add A a i* (− *A* \$ *a* \$ *j*) \$ *a* \$ *b*
unfolding *rw row-add-def* **apply** *simp*
unfolding *A′-def rw1 rw2 rw3* **..**
qed
qed
qed

lemma *row-add-iterate-eq-Gauss-Jordan-in-ij*:
fixes *A*::′*a*::{*field*} ^′*n* ^′*m*::{*mod-type*} **and** *i*::′*m* **and** *j*::′*n*
defines *A′*: *A′*== *mult-row* (*interchange-rows A i* (*LEAST n. A* \$ *n* \$ *j* ≠ *0* ∧ *i* ≤ *n*)) *i* (*1* / (*interchange-rows A i* (*LEAST n. A* \$ *n* \$ *j* ≠ *0* ∧ *i* ≤ *n*)) \$ *i* \$ *j*)
shows *row-add-iterate A′* (*nrows A* − *1*) *i j* = *Gauss-Jordan-in-ij A i j*
proof (*unfold Gauss-Jordan-in-ij-def Let-def*, *vector*, *auto*)
fix *ia*
have *interchange-rw*: *A* \$ (*LEAST n. A* \$ *n* \$ *j* ≠ *0* ∧ *i* ≤ *n*) \$ *j* = *interchange-rows A i* (*LEAST n. A* \$ *n* \$ *j* ≠ *0* ∧ *i* ≤ *n*) \$ *i* \$ *j*
using *interchange-rows-j*[*symmetric*, *of A* (*LEAST n. A* \$ *n* \$ *j* ≠ *0* ∧ *i* ≤ *n*)] **by** *auto*
show *row-add-iterate A′* (*nrows A* − *Suc 0*) *i j* \$ *i* \$ *ia* =
mult-row (*interchange-rows A i* (*LEAST n. A* \$ *n* \$ *j* ≠ *0* ∧ *i* ≤ *n*)) *i* (*1* / *A* \$ (*LEAST n. A* \$ *n* \$ *j* ≠ *0* ∧ *i* ≤ *n*) \$ *j*) \$ *i* \$ *ia*

unfolding *interchange-rw* **unfolding** A'
proof (*rule row-add-iterate-preserves-pivot-row*, *unfold nrows-def*)
show $CARD('m) - Suc\ 0 < CARD('m)$ **by** *simp*
have *to-nat i* $<$ $CARD('m)$ **using** *bij-to-nat*[**where** $?'a='m$] **unfolding** *bij-betw-def* **by** *auto*
thus *to-nat i* $\leq CARD('m) - Suc\ 0$ **by** *auto*
qed
next
fix *ia iaa*
have *interchange-rw*: $A\ \$\ (LEAST\ n.\ A\ \$\ n\ \$\ j \neq 0 \wedge i \leq n)\ \$\ j$ = *interchange-rows* $A\ i\ (LEAST\ n.\ A\ \$\ n\ \$\ j \neq 0 \wedge i \leq n)\ \$\ i\ \$\ j$
using *interchange-rows-j*[*symmetric*, *of A* $(LEAST\ n.\ A\ \$\ n\ \$\ j \neq 0 \wedge i \leq n)$] **by** *auto*
assume *ia-not-i*: $ia \neq i$
have *rw*: ($-$ *interchange-rows* $A\ i\ (LEAST\ n.\ A\ \$\ n\ \$\ j \neq 0 \wedge i \leq n)\ \$\ ia\ \$\ j$)
= $-$ *mult-row* (*interchange-rows* $A\ i\ (LEAST\ n.\ A\ \$\ n\ \$\ j \neq 0 \wedge i \leq n)$) i (1 / *interchange-rows* $A\ i\ (LEAST\ n.\ A\ \$\ n\ \$\ j \neq 0 \wedge i \leq n)\ \$\ i\ \$\ j$) $\$\ ia\ \$\ j$
unfolding *interchange-rows-def mult-row-def* **using** *ia-not-i* **by** *auto*
show *row-add-iterate* A' (*nrows* $A - Suc\ 0$) $i\ j\ \$\ ia\ \$\ iaa$ =
row-add (*mult-row* (*interchange-rows* $A\ i\ (LEAST\ n.\ A\ \$\ n\ \$\ j \neq 0 \wedge i \leq n)$) i (1 / $A\ \$\ (LEAST\ n.\ A\ \$\ n\ \$\ j \neq 0 \wedge i \leq n)\ \$\ j$)) *ia i*
($-$ *interchange-rows* $A\ i\ (LEAST\ n.\ A\ \$\ n\ \$\ j \neq 0 \wedge i \leq n)\ \$\ ia\ \$\ j$) $\$$
ia $\$$
iaa
unfolding *interchange-rw* A' *rw*
proof (*rule row-add-iterate-eq-row-add*[*of ia i* (*nrows* $A - Suc\ 0$) - *j iaa*], *unfold nrows-def*)
show $ia \neq i$ **using** *ia-not-i* **.**
show $CARD('m) - Suc\ 0 < CARD('m)$ **by** *simp*
have *to-nat ia* $<$ $CARD('m)$ **using** *bij-to-nat*[**where** $?'a='m$] **unfolding** *bij-betw-def* **by** *auto*
thus *to-nat ia* $\leq CARD('m) - Suc\ 0$ **by** *simp*
qed
qed

lemma *invertible-Gauss-Jordan-column-k*:
fixes A::$'a$::{*field*} ^ $'n$::{*mod-type*} ^ $'m$::{*mod-type*} **and** k::*nat*
shows $\exists P.$ *invertible* $P \wedge$ (*snd* (*Gauss-Jordan-column-k* (i,A) k)) = $P{*}{*}A$
unfolding *Gauss-Jordan-column-k-def Let-def*
proof (*auto*)
show $\exists P.$ *invertible* $P \wedge A = P ** A$ **and** $\exists P.$ *invertible* $P \wedge A = P ** A$ **using** *invertible-mat-1 matrix-mul-lid*[*of A*] **by** *auto*
next
fix m
assume i: $i \neq$ *nrows* A
and *i-le-m*: *from-nat* $i \leq m$ **and** *Amk-not-zero*: $A\ \$\ m\ \$$ *from-nat* $k \neq 0$
def *A-interchange* $\equiv$ (*interchange-rows* A (*from-nat* i) $(LEAST\ n.\ A\ \$\ n\ \$$

from-nat k ≠ 0 ∧ (from-nat i) ≤ n))
def *A-mult ≡ (mult-row A-interchange (from-nat i) (1 / (A-interchange $ (from-nat i) $ from-nat k)))*
obtain *P* **where** *inv-P*: *invertible P* **and** *PA*: *A-interchange = P**A*
unfolding *A-interchange-def*
using *interchange-rows-mat-1[of from-nat i (LEAST n. A $ n $ from-nat k ≠ 0 ∧ from-nat i ≤ n) A]*
using *invertible-interchange-rows[of from-nat i (LEAST n. A $ n $ from-nat k ≠ 0 ∧ from-nat i ≤ n)]*
by *fastforce*
def *Q ≡ (mult-row (mat 1) (from-nat i) (1 / (A-interchange $ (from-nat i) $ from-nat k)))::'a^'m::{mod-type}^'m::{mod-type}*
have *Q-A-interchange*: *A-mult = Q**A-interchange* **unfolding** *A-mult-def A-interchange-def Q-def* **unfolding** *mult-row-mat-1* **..**
have *inv-Q*: *invertible Q*
proof *(unfold Q-def, rule invertible-mult-row′, unfold A-interchange-def, rule LeastI2-ex)*
show *∃a. A $ a $ from-nat k ≠ 0 ∧ (from-nat i) ≤ a* **using** *i-le-m Amk-not-zero* **by** *blast*
show *⋀x. A $ x $ from-nat k ≠ 0 ∧ (from-nat i) ≤ x ⟹ 1 / interchange-rows A (from-nat i) x $ (from-nat i) $ from-nat k ≠ 0*
using *interchange-rows-i mult-zero-left nonzero-divide-eq-eq zero-neq-one* **by** *fastforce*
qed
obtain *Pa* **where** *inv-Pa*: *invertible Pa* **and** *Pa*: *row-add-iterate (Q ** (P ** A)) (nrows A − 1) (from-nat i) (from-nat k) = Pa ** (Q ** (P ** A))*
using *invertible-row-add-iterate* **by** *(metis (full-types) diff-less nrows-def zero-less-card-finite zero-less-one)*
show *∃P. invertible P ∧ Gauss-Jordan-in-ij A (from-nat i) (from-nat k) = P ** A*
proof *(rule exI[of - Pa**Q**P], rule conjI)*
show *invertible (Pa ** Q ** P)* **using** *inv-P inv-Pa inv-Q invertible-mult* **by** *auto*
have *Gauss-Jordan-in-ij A (from-nat i) (from-nat k) = row-add-iterate A-mult (nrows A − 1) (from-nat i) (from-nat k)*
unfolding *row-add-iterate-eq-Gauss-Jordan-in-ij[symmetric] A-mult-def A-interchange-def* **..**
also have ... *= Pa ** (Q ** (P ** A))* **using** *Pa* **unfolding** *PA[symmetric] Q-A-interchange[symmetric]* **.**
also have ... *= Pa ** Q ** P ** A* **unfolding** *matrix-mul-assoc* **..**
finally show *Gauss-Jordan-in-ij A (from-nat i) (from-nat k) = Pa ** Q ** P ** A* **.**
qed
qed

lemma *invertible-Gauss-Jordan-up-to-k*:
fixes *A::'a::{field}^'n::{mod-type}^'m::{mod-type}*
shows *∃P. invertible P ∧ (Gauss-Jordan-upt-k A k) = P**A*

proof (*induct k*)
  case *0*
  have *rw*: [*0*..<*Suc 0*] = [*0*] **by** *fastforce*
  show *?case*
    unfolding *Gauss-Jordan-upt-k-def rw foldl.simps*
    using *invertible-Gauss-Jordan-column-k* **.**
  case (*Suc k*)
  have *rw2*: [*0*..<*Suc* (*Suc k*)] = [*0*..< *Suc k*] @ [(*Suc k*)] **by** *simp*
  obtain P' **where** *inv-P'*: *invertible* P' **and** *Gk-eq-P'A*: *Gauss-Jordan-upt-k A k* = P' ** *A* **using** *Suc.hyps* **by** *force*
  have *g*: *Gauss-Jordan-upt-k A k* = *snd* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc k*]) **unfolding** *Gauss-Jordan-upt-k-def* **by** *auto*
  show *?case* **unfolding** *Gauss-Jordan-upt-k-def* **unfolding** *rw2 foldl-append foldl.simps*
    apply (*subst pair-collapse*[*symmetric*, *of* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc k*]), *unfolded g*[*symmetric*]])
    using *invertible-Gauss-Jordan-column-k*
    using *Suc.hyps* **using** *invertible-mult matrix-mul-assoc* **by** *metis*
qed

lemma *inj-index-independent-rows*:
  fixes *A*::*real*^'*m*::{*mod-type*}^'*n*::{*finite*,*one*,*plus*,*ord*}
  assumes *rref-A*: *reduced-row-echelon-form A*
  and *x*: *row x A* ∈ {*row i A* |*i*. *row i A* ≠ *0*}
  and *eq*: *A* $ *x* = *A* $ *y*
  shows *x* = *y*
proof (*rule ccontr*)
  assume *x-not-y*: *x* ≠ *y*
  have *not-zero-x*: ¬ *is-zero-row x A* **using** *x* **unfolding** *is-zero-row-def* **unfolding** *is-zero-row-upt-k-def* **unfolding** *row-def vec-eq-iff ncols-def* **by** *auto*
  hence *not-zero-y*: ¬ *is-zero-row y A* **using** *eq* **unfolding** *is-zero-row-def'* **by** *simp*
  have *Ax*: *A* $ *x* $ (*LEAST k*. *A* $ *x* $ *k* ≠ *0*) = *1* **using** *not-zero-x rref-condition2*[*OF rref-A*] **by** *simp*
  have *Ay*: *A* $ *x* $ (*LEAST k*. *A* $ *y* $ *k* ≠ *0*) = *0* **using** *not-zero-y x-not-y rref-condition4*[*OF rref-A*] **by** *fast*
  show *False* **using** *Ax Ay* **unfolding** *eq* **by** *simp*
qed

The final results

lemma *invertible-Gauss-Jordan*:
  fixes *A*::'*a*::{*field*}^'*n*::{*mod-type*}^'*m*::{*mod-type*}
  shows ∃ *P*. *invertible P* ∧ (*Gauss-Jordan A*) = *P****A* **unfolding** *Gauss-Jordan-def* **using** *invertible-Gauss-Jordan-up-to-k* **.**

lemma *rank-Gauss-Jordan*:
  fixes *A*::*real*^'*n*::{*mod-type*}^'*m*::{*mod-type*}
  shows *rank A* = *rank* (*Gauss-Jordan A*) **by** (*metis invertible-Gauss-Jordan invertible-matrix-mult-left-rank*)

4.5 Lemmas for code generation

lemma [*code abstract*]:
shows *vec-nth* (*Gauss-Jordan-in-ij A i j*) = (*let n* = (*LEAST n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*);
interchange-A = (*interchange-rows A i n*);
A′ = *mult-row interchange-A i* (*1/interchange-A*$*i*$*j*) *in*
(% *s. if s=i then A′* $ *s else* (*row-add A′ s i* (−(*interchange-A*$*s*$*j*))) $ *s*))
unfolding *Gauss-Jordan-in-ij-def Let-def* **by** *fastforce*

lemma *rank-Gauss-Jordan-code*[*code*]:
fixes *A*::*real*^′*n*::{*mod-type*}^′*m*::{*mod-type*}
shows *rank A* = *card* {*row i* (*Gauss-Jordan A*) |*i. row i* (*Gauss-Jordan A*) ≠ *0*}
by (*metis* (*mono-tags*) *rank-Gauss-Jordan rref-Gauss-Jordan rref-rank*)

lemma *dim-null-space*[*code-unfold*]:
fixes *A*::*real*^′*a*^′*b*
shows *dim* (*null-space A*) = *DIM* (*real*^′*a*) − *rank* (*A*)
apply (*rule add-implies-diff*)
using *rank-nullity-theorem-matrices*
unfolding *col-space-eq*[*symmetric*] *rank-eq-dim-col-space*[*symmetric*] **..**

lemma *rank-eq-dim-col-space′*[*code-unfold*]:
dim (*Dim-Formula.col-space A*) = *rank A*
by (*metis col-space-eq rank-eq-dim-col-space*)

lemma [*code abstract*]: *vec-nth* (*row i A*) = (*op* $ (*A* $ *i*)) **unfolding** *row-def* **by** *auto*
lemmas *rank-col-rank*[*symmetric*, *code-unfold*]
lemmas *rank-def*[*symmetric*, *code-unfold*]
lemmas *row-rank-def*[*symmetric*, *code-unfold*]
lemmas *col-rank-def*[*symmetric*, *code-unfold*]
lemmas *DIM-cart*[*code-unfold*]

end

theory *Code-Set*
imports *Main*
begin

The following setup could help to get code generation for List.coset, but it neither works correctly it complains that code equations for remove are missed, even when List.coset should be rewritten to an enum:

declare *union-coset-filter* [*code del*]
declare *minus-coset-filter* [*code del*]
declare *inter-coset-fold* [*code del*]
declare *remove-code*(*2*) [*code del*]

declare *insert-code(2)* [*code del*]

code-datatype *set*

The following code equation does never work, because the minus for sets requires Set.remove, which has been deleted from the code generation setup:

lemma [*code*]: *List.coset* (*l*::*'a*::*enum list*) = *set* (*enum-class.enum*) − *set l*
 by (*metis Compl-eq-Diff-UNIV coset-def enum-UNIV*)

Now the following examples work:

value [*code*] *UNIV*::*bool set*
value [*code*] *List.coset* ([]::*bool list*)
value [*code*] *UNIV*::*Enum.finite-2 set*
value [*code*] *List.coset* ([]::*Enum.finite-2 list*)
value [*code*] *List.coset* ([]::*Enum.finite-5 list*)

This declare allows to calculate the cardinal of the non-zero rows set.

declare *compl-coset*[*code del*]

end

theory *Gauss-Jordan-Examples*
imports
 Gauss-Jordan
 Code-Set
begin

Definitions to transform a matrix to a list of list and vice versa

definition *vec-to-list* :: *'a^'n*::{*finite*, *enum*} => *'a list*
 where *vec-to-list A* = *map* (*op* $ *A*) (*enum-class.enum*::*'n list*)

definition *matrix-to-list-of-list* :: *'a^'n*::{*finite*, *enum*}*^'m*::{*finite*, *enum*} => *'a list list*
 where *matrix-to-list-of-list A* = *map* (*vec-to-list*) (*map* (*op* $ *A*) (*enum-class.enum*::*'m list*))

This definition should be equivalent to *vector ?l* = (χ*i*. *foldr* (λ*x f n*. (*f* (*n* + (*1*::*?'b*)))(*n* := *x*)) *?l* (λ*n x*. *0*::*?'a*) (*1*::*?'b*) *i*) (in suitable types)

definition *list-to-vec* :: *'a list* => *'a^'n*::{*finite*, *enum*, *mod-type*}
 where *list-to-vec xs* = *vec-lambda* (% *i*. *xs* ! (*to-nat i*))

lemma [*code abstract*]: *vec-nth* (*list-to-vec xs*) = (%*i*. *xs* ! (*to-nat i*))
 unfolding *list-to-vec-def* **by** *fastforce*

definition *list-of-list-to-matrix* :: *'a list list* => *'a^'n*::{*finite*, *enum*, *mod-type*}*^'m*::{*finite*, *enum*, *mod-type*}
 where *list-of-list-to-matrix xs* = *vec-lambda* (%*i*. *list-to-vec* (*xs* ! (*to-nat i*)))

lemma [*code abstract*]: *vec-nth* (*list-of-list-to-matrix xs*) = (%*i*. *list-to-vec* (*xs* ! (*to-nat i*)))
  unfolding *list-of-list-to-matrix-def* **by** *auto*

end

5 Implementation of natural numbers as binary numerals

theory *Code-Binary-Nat*
imports *Main*
begin

When generating code for functions on natural numbers, the canonical representation using *0* and *Suc* is unsuitable for computations involving large numbers. This theory refines the representation of natural numbers for code generation to use binary numerals, which do not grow linear in size but logarithmic.

5.1 Representation

code-datatype *0*::*nat nat-of-num*

lemma [*code-abbrev*]:
  nat-of-num = *numeral*
  by (*fact nat-of-num-numeral*)

lemma [*code*]:
  num-of-nat 0 = *Num.One*
  num-of-nat (*nat-of-num k*) = *k*
  by (*simp-all add*: *nat-of-num-inverse*)

lemma [*code*]:
  (*1*::*nat*) = *Numeral1*
  by *simp*

lemma [*code-abbrev*]: *Numeral1* = (*1*::*nat*)
  by *simp*

lemma [*code*]:
  Suc n = *n* + *1*
  by *simp*

5.2 Basic arithmetic

lemma [*code*, *code del*]:
  (*plus* :: *nat* ⇒ -) = *plus* **..**

lemma *plus-nat-code* [*code*]:
  nat-of-num k + *nat-of-num l* = *nat-of-num* (*k* + *l*)
  m + *0* = (*m*::*nat*)
  0 + *n* = (*n*::*nat*)
  by (*simp-all add*: *nat-of-num-numeral*)

Bounded subtraction needs some auxiliary

definition *dup* :: *nat* ⇒ *nat* **where**
  dup n = *n* + *n*

lemma *dup-code* [*code*]:
  dup 0 = *0*
  dup (*nat-of-num k*) = *nat-of-num* (*Num.Bit0 k*)
  by (*simp-all add*: *dup-def numeral-Bit0*)

definition *sub* :: *num* ⇒ *num* ⇒ *nat option* **where**
  sub k l = (*if k* ≥ *l then Some* (*numeral k* − *numeral l*) *else None*)

lemma *sub-code* [*code*]:
  sub Num.One Num.One = *Some 0*
  sub (*Num.Bit0 m*) *Num.One* = *Some* (*nat-of-num* (*Num.BitM m*))
  sub (*Num.Bit1 m*) *Num.One* = *Some* (*nat-of-num* (*Num.Bit0 m*))
  sub Num.One (*Num.Bit0 n*) = *None*
  sub Num.One (*Num.Bit1 n*) = *None*
  sub (*Num.Bit0 m*) (*Num.Bit0 n*) = *Option.map dup* (*sub m n*)
  sub (*Num.Bit1 m*) (*Num.Bit1 n*) = *Option.map dup* (*sub m n*)
  sub (*Num.Bit1 m*) (*Num.Bit0 n*) = *Option.map* (λ*q*. *dup q* + *1*) (*sub m n*)
  sub (*Num.Bit0 m*) (*Num.Bit1 n*) = (*case sub m n of None* ⇒ *None*
    | *Some q* ⇒ *if q* = *0 then None else Some* (*dup q* − *1*))
  apply (*auto simp add*: *nat-of-num-numeral*
    Num.dbl-def Num.dbl-inc-def Num.dbl-dec-def
    Let-def le-imp-diff-is-add BitM-plus-one sub-def dup-def)
  apply (*simp-all add*: *sub-non-positive*)
  apply (*simp-all add*: *sub-non-negative* [*symmetric*, **where** *?'a* = *int*])
  done

lemma [*code*, *code del*]:
  (*minus* :: *nat* ⇒ -) = *minus* **..**

lemma *minus-nat-code* [*code*]:
  nat-of-num k − *nat-of-num l* = (*case sub k l of None* ⇒ *0* | *Some j* ⇒ *j*)
  m − *0* = (*m*::*nat*)
  0 − *n* = (*0*::*nat*)
  by (*simp-all add*: *nat-of-num-numeral sub-non-positive sub-def*)

lemma [*code*, *code del*]:
(*times* :: *nat* ⇒ -) = *times* **..**

lemma *times-nat-code* [*code*]:
nat-of-num k ∗ *nat-of-num l* = *nat-of-num* (*k* ∗ *l*)
m ∗ *0* = (*0*::*nat*)
0 ∗ *n* = (*0*::*nat*)
by (*simp-all add*: *nat-of-num-numeral*)

lemma [*code*, *code del*]:
(*HOL.equal* :: *nat* ⇒ -) = *HOL.equal* **..**

lemma *equal-nat-code* [*code*]:
HOL.equal 0 (*0*::*nat*) ⟷ *True*
HOL.equal 0 (*nat-of-num l*) ⟷ *False*
HOL.equal (*nat-of-num k*) *0* ⟷ *False*
HOL.equal (*nat-of-num k*) (*nat-of-num l*) ⟷ *HOL.equal k l*
by (*simp-all add*: *nat-of-num-numeral equal*)

lemma *equal-nat-refl* [*code nbe*]:
HOL.equal (*n*::*nat*) *n* ⟷ *True*
by (*rule equal-refl*)

lemma [*code*, *code del*]:
(*less-eq* :: *nat* ⇒ -) = *less-eq* **..**

lemma *less-eq-nat-code* [*code*]:
0 ≤ (*n*::*nat*) ⟷ *True*
nat-of-num k ≤ *0* ⟷ *False*
nat-of-num k ≤ *nat-of-num l* ⟷ *k* ≤ *l*
by (*simp-all add*: *nat-of-num-numeral*)

lemma [*code*, *code del*]:
(*less* :: *nat* ⇒ -) = *less* **..**

lemma *less-nat-code* [*code*]:
(*m*::*nat*) < *0* ⟷ *False*
0 < *nat-of-num l* ⟷ *True*
nat-of-num k < *nat-of-num l* ⟷ *k* < *l*
by (*simp-all add*: *nat-of-num-numeral*)

5.3 Conversions

lemma [*code*, *code del*]:
of-nat = *of-nat* **..**

lemma *of-nat-code* [*code*]:
of-nat 0 = *0*
of-nat (*nat-of-num k*) = *numeral k*

by (*simp-all add*: *nat-of-num-numeral*)

5.4 Case analysis

Case analysis on natural numbers is rephrased using a conditional expression:

lemma [*code*, *code-unfold*]:
nat-case = ($\lambda f\ g\ n$. *if* $n = 0$ *then* f *else* g ($n - 1$))
by (*auto simp add*: *fun-eq-iff dest*!: *gr0-implies-Suc*)

5.5 Preprocessors

The term *Suc n* is no longer a valid pattern. Therefore, all occurrences of this term in a position where a pattern is expected (i.e. on the left-hand side of a recursion equation) must be eliminated. This can be accomplished by applying the following transformation rules:

lemma *Suc-if-eq*: ($\bigwedge n.\ f\ (Suc\ n) \equiv h\ n) \Longrightarrow f\ 0 \equiv g \Longrightarrow$
$f\ n \equiv$ *if* $n = 0$ *then* g *else* $h\ (n - 1)$
by (*rule eq-reflection*) (*cases n*, *simp-all*)

The rules above are built into a preprocessor that is plugged into the code generator. Since the preprocessor for introduction rules does not know anything about modes, some of the modes that worked for the canonical representation of natural numbers may no longer work.

code-modulename *SML*
Code-Binary-Nat Arith

code-modulename *OCaml*
Code-Binary-Nat Arith

code-modulename *Haskell*
Code-Binary-Nat Arith

hide-const (**open**) *dup sub*

end

theory *Code-Natural*
imports ../*Main*
begin

6 Alternative representation of *code-numeral* for *Haskell* and *Scala*

code-include *Haskell Natural*

```
⟪import Data.Array.ST;

newtype Natural = Natural Integer deriving (Eq, Show, Read);

instance Num Natural where {
  fromInteger k = Natural (if k >= 0 then k else 0);
  Natural n + Natural m = Natural (n + m);
  Natural n − Natural m = fromInteger (n − m);
  Natural n ∗ Natural m = Natural (n ∗ m);
  abs n = n;
  signum - = 1;
  negate n = error negate Natural;
};

instance Ord Natural where {
  Natural n <= Natural m = n <= m;
  Natural n < Natural m = n < m;
};

instance Ix Natural where {
  range (Natural n, Natural m) = map Natural (range (n, m));
  index (Natural n, Natural m) (Natural q) = index (n, m) q;
  inRange (Natural n, Natural m) (Natural q) = inRange (n, m) q;
  rangeSize (Natural n, Natural m) = rangeSize (n, m);
};

instance Real Natural where {
  toRational (Natural n) = toRational n;
};

instance Enum Natural where {
  toEnum k = fromInteger (toEnum k);
  fromEnum (Natural n) = fromEnum n;
};

instance Integral Natural where {
  toInteger (Natural n) = n;
  divMod n m = quotRem n m;
  quotRem (Natural n) (Natural m)
    | (m == 0) = (0, Natural n)
    | otherwise = (Natural k, Natural l) where (k, l) = quotRem n m;
};⟫
```

code-reserved *Haskell Natural*

code-include *Scala Natural*

```
⟪object Natural {

  def apply(numeral: BigInt): Natural = new Natural(numeral max 0)
  def apply(numeral: Int): Natural = Natural(BigInt(numeral))
  def apply(numeral: String): Natural = Natural(BigInt(numeral))

}

class Natural private(private val value: BigInt) {

  override def hashCode(): Int = this.value.hashCode()

  override def equals(that: Any): Boolean = that match {
    case that: Natural => this equals that
    case _ => false
  }

  override def toString(): String = this.value.toString

  def equals(that: Natural): Boolean = this.value == that.value

  def as_BigInt: BigInt = this.value
  def as_Int: Int = if (this.value >= scala.Int.MinValue && this.value <= scala.Int.MaxValue)
      this.value.intValue
    else error(Int value out of range:  + this.value.toString)

  def +(that: Natural): Natural = new Natural(this.value + that.value)
  def −(that: Natural): Natural = Natural(this.value − that.value)
  def ∗(that: Natural): Natural = new Natural(this.value ∗ that.value)

  def /%(that: Natural): (Natural, Natural) = if (that.value == 0) (new Natural(0), this)
    else {
      val (k, l) = this.value /% that.value
      (new Natural(k), new Natural(l))
    }

  def <=(that: Natural): Boolean = this.value <= that.value

  def <(that: Natural): Boolean = this.value < that.value

}
⟫
```

code-reserved *Scala Natural*

code-type *code-numeral*

(*Haskell Natural.Natural*)
(*Scala Natural*)

setup ⟪
fold (*Numeral.add-code* @{*const-name Code-Numeral.Num*}
false Code-Printer.literal-alternative-numeral) [*Haskell*, *Scala*]
⟫

code-instance *code-numeral* :: *equal*
(*Haskell* −)

code-const *0*::*code-numeral*
(*Haskell 0*)
(*Scala Natural*(*0*))

code-const *plus* :: *code-numeral* ⇒ *code-numeral* ⇒ *code-numeral*
(*Haskell* **infixl** *6* +)
(*Scala* **infixl** *7* +)

code-const *minus* :: *code-numeral* ⇒ *code-numeral* ⇒ *code-numeral*
(*Haskell* **infixl** *6* −)
(*Scala* **infixl** *7* −)

code-const *times* :: *code-numeral* ⇒ *code-numeral* ⇒ *code-numeral*
(*Haskell* **infixl** *7* ∗)
(*Scala* **infixl** *8* ∗)

code-const *Code-Numeral.div-mod*
(*Haskell divMod*)
(*Scala* **infixl** *8* /%)

code-const *HOL.equal* :: *code-numeral* ⇒ *code-numeral* ⇒ *bool*
(*Haskell* **infix** *4* ==)
(*Scala* **infixl** *5* ==)

code-const *less-eq* :: *code-numeral* ⇒ *code-numeral* ⇒ *bool*
(*Haskell* **infix** *4* <=)
(*Scala* **infixl** *4* <=)

code-const *less* :: *code-numeral* ⇒ *code-numeral* ⇒ *bool*
(*Haskell* **infix** *4* <)
(*Scala* **infixl** *4* <)

end

7 Pretty integer literals for code generation

theory *Code-Integer*

imports *Main Code-Natural*
begin

Representation-ignorant code equations for conversions.

lemma *nat-code* [*code*]:
  nat k = (*if k* $\leq$ *0 then 0 else*
     let
       (*l*, *j*) = *divmod-int k 2*;
       n = *nat l*;
       l' = *n* + *n*
     in if j = *0 then l' else Suc l'*)
proof −
  have *2* = *nat 2* **by** *simp*
  show *?thesis*
    apply (*subst mult-2* [*symmetric*])
    apply (*auto simp add*: *Let-def divmod-int-mod-div not-le*
     nat-div-distrib nat-mult-distrib mult-div-cancel mod-2-not-eq-zero-eq-one-int)
    apply (*unfold* ‹*2* = *nat 2*›)
    apply (*subst nat-mod-distrib* [*symmetric*])
    apply *simp-all*
  done
qed

lemma (**in** *ring-1*) *of-int-code*:
  of-int k = (*if k* = *0 then 0*
     else if k < *0 then* − *of-int* (− *k*)
     else let
       (*l*, *j*) = *divmod-int k 2*;
       l' = *2* ∗ *of-int l*
     in if j = *0 then l' else l'* + *1*)
proof −
  from *mod-div-equality* **have** ∗: *of-int k* = *of-int* (*k div 2* ∗ *2* + *k mod 2*) **by** *simp*
  show *?thesis*
    by (*simp add*: *Let-def divmod-int-mod-div mod-2-not-eq-zero-eq-one-int*
      of-int-add [*symmetric*]) (*simp add*: ∗ *mult-commute*)
qed

declare *of-int-code* [*code*]

HOL numeral expressions are mapped to integer literals in target languages, using predefined target language operations for abstract integer operations.

code-type *int*
  (*SML IntInf.int*)
  (*OCaml Big'-int.big'-int*)
  (*Haskell Integer*)
  (*Scala BigInt*)
  (*Eval int*)

```
code-instance int :: equal
  (Haskell -)

code-const 0::int
  (SML 0)
  (OCaml Big'-int.zero'-big'-int)
  (Haskell 0)
  (Scala BigInt(0))

setup ⟪
  fold (Numeral.add-code @{const-name Int.Pos}
    false Code-Printer.literal-numeral) [SML, OCaml, Haskell, Scala]
⟫

setup ⟪
  fold (Numeral.add-code @{const-name Int.Neg}
    true Code-Printer.literal-numeral) [SML, OCaml, Haskell, Scala]
⟫

code-const op + :: int ⇒ int ⇒ int
  (SML IntInf.+ ((-), (-)))
  (OCaml Big'-int.add'-big'-int)
  (Haskell infixl 6 +)
  (Scala infixl 7 +)
  (Eval infixl 8 +)

code-const uminus :: int ⇒ int
  (SML IntInf.~)
  (OCaml Big'-int.minus'-big'-int)
  (Haskell negate)
  (Scala !(- -))
  (Eval ~/ -)

code-const op - :: int ⇒ int ⇒ int
  (SML IntInf.- ((-), (-)))
  (OCaml Big'-int.sub'-big'-int)
  (Haskell infixl 6 -)
  (Scala infixl 7 -)
  (Eval infixl 8 -)

code-const Int.dup
  (SML IntInf.*/ (2,/ (-)))
  (OCaml Big'-int.mult'-big'-int/ 2)
  (Haskell !(2 * -))
  (Scala !(2 * -))
  (Eval !(2 * -))

code-const Int.sub
  (SML !(raise/ Fail/ sub))
```

(*OCaml failwith/ sub*)
(*Haskell error/ sub*)
(*Scala* !*sys*.*error*(*sub*))

code-const *op* ∗ :: *int* ⇒ *int* ⇒ *int*
(*SML IntInf*.∗ ((-), (-)))
(*OCaml Big′-int*.*mult′-big′-int*)
(*Haskell* **infixl** *7* ∗)
(*Scala* **infixl** *8* ∗)
(*Eval* **infixl** *9* ∗)

code-const *pdivmod*
(*SML IntInf*.*divMod*/ (*IntInf*.*abs* -,/ *IntInf*.*abs* -))
(*OCaml Big′-int*.*quomod′-big′-int*/ (*Big′-int*.*abs′-big′-int* -)/ (*Big′-int*.*abs′-big′-int* -))
(*Haskell divMod*/ (*abs* -)/ (*abs* -))
(*Scala* !((*k*: *BigInt*) => (*l*: *BigInt*) =>/ *if* (*l* == *0*)/ (*BigInt*(*0*), *k*) *else*/ (*k*.*abs* ′/% *l*.*abs*)))
(*Eval Integer*.*div′-mod*/ (*abs* -)/ (*abs* -))

code-const *HOL*.*equal* :: *int* ⇒ *int* ⇒ *bool*
(*SML* !((- : *IntInf*.*int*) = -))
(*OCaml Big′-int*.*eq′-big′-int*)
(*Haskell* **infix** *4* ==)
(*Scala* **infixl** *5* ==)
(*Eval* **infixl** *6* =)

code-const *op* ≤ :: *int* ⇒ *int* ⇒ *bool*
(*SML IntInf*.<= ((-), (-)))
(*OCaml Big′-int*.*le′-big′-int*)
(*Haskell* **infix** *4* <=)
(*Scala* **infixl** *4* <=)
(*Eval* **infixl** *6* <=)

code-const *op* < :: *int* ⇒ *int* ⇒ *bool*
(*SML IntInf*.< ((-), (-)))
(*OCaml Big′-int*.*lt′-big′-int*)
(*Haskell* **infix** *4* <)
(*Scala* **infixl** *4* <)
(*Eval* **infixl** *6* <)

code-const *Code-Numeral*.*int-of*
(*SML IntInf*.*fromInt*)
(*OCaml* -)
(*Haskell Prelude*.*toInteger*)
(*Scala* !-.*as′-BigInt*)
(*Eval* -)

code-const *Code-Evaluation*.*term-of* :: *int* ⇒ *term*

(*Eval HOLogic.mk'-number/ HOLogic.intT*)

end

8 Implementation of natural numbers by target-language integers

theory *Efficient-Nat*
imports *Code-Binary-Nat Code-Integer Main*
begin

The efficiency of the generated code for natural numbers can be improved drastically by implementing natural numbers by target-language integers. To do this, just include this theory.

8.1 Target language fundamentals

For ML, we map *nat* to target language integers, where we ensure that values are always non-negative.

code-type *nat*
(*SML IntInf.int*)
(*OCaml Big'-int.big'-int*)
(*Eval int*)

For Haskell and Scala we define our own *nat* type. The reason is that we have to distinguish type class instances for *nat* and *int*.

code-include *Haskell Nat*
```
⟨⟨newtype Nat = Nat Integer deriving (Eq, Show, Read);

instance Num Nat where {
  fromInteger k = Nat (if k >= 0 then k else 0);
  Nat n + Nat m = Nat (n + m);
  Nat n − Nat m = fromInteger (n − m);
  Nat n ∗ Nat m = Nat (n ∗ m);
  abs n = n;
  signum - = 1;
  negate n = error negate Nat;
};

instance Ord Nat where {
  Nat n <= Nat m = n <= m;
  Nat n < Nat m = n < m;
};

instance Real Nat where {
  toRational (Nat n) = toRational n;
```

```
};

instance Enum Nat where {
  toEnum k = fromInteger (toEnum k);
  fromEnum (Nat n) = fromEnum n;
};

instance Integral Nat where {
  toInteger (Nat n) = n;
  divMod n m = quotRem n m;
  quotRem (Nat n) (Nat m)
    | (m == 0) = (0, Nat n)
    | otherwise = (Nat k, Nat l) where (k, l) = quotRem n m;
};
⟫
```

code-reserved *Haskell Nat*

code-include *Scala Nat*

```
⟪object Nat {

  def apply(numeral: BigInt): Nat = new Nat(numeral max 0)
  def apply(numeral: Int): Nat = Nat(BigInt(numeral))
  def apply(numeral: String): Nat = Nat(BigInt(numeral))

}

class Nat private(private val value: BigInt) {

  override def hashCode(): Int = this.value.hashCode()

  override def equals(that: Any): Boolean = that match {
    case that: Nat => this equals that
    case - => false
  }

  override def toString(): String = this.value.toString

  def equals(that: Nat): Boolean = this.value == that.value

  def as-BigInt: BigInt = this.value
  def as-Int: Int = if (this.value >= scala.Int.MinValue && this.value <= scala.Int.MaxValue)
      this.value.intValue
    else error(Int value out of range: + this.value.toString)

  def +(that: Nat): Nat = new Nat(this.value + that.value)
  def −(that: Nat): Nat = Nat(this.value − that.value)
  def ∗(that: Nat): Nat = new Nat(this.value ∗ that.value)
```

```
  def /%(that: Nat): (Nat, Nat) = if (that.value == 0) (new Nat(0), this)
    else {
      val (k, l) = this.value /% that.value
      (new Nat(k), new Nat(l))
    }

  def <=(that: Nat): Boolean = this.value <= that.value

  def <(that: Nat): Boolean = this.value < that.value

}
»
```

code-reserved *Scala Nat*

code-type *nat*
  (*Haskell Nat.Nat*)
  (*Scala Nat*)

code-instance *nat* :: *equal*
  (*Haskell* −)

setup «
  fold (*Numeral.add-code* @{*const-name nat-of-num*}
    false Code-Printer.literal-positive-numeral) [*SML*, *OCaml*, *Haskell*, *Scala*]
»

code-const *0::nat*
  (*SML 0*)
  (*OCaml Big'-int.zero'-big'-int*)
  (*Haskell 0*)
  (*Scala Nat(0)*)

8.2 Conversions

Since natural numbers are implemented using integers in ML, the coercion function *int* of type *nat* ⇒ *int* is simply implemented by the identity function. For the *nat* function for converting an integer to a natural number, we give a specific implementation using an ML expression that returns its input value, provided that it is non-negative, and otherwise returns *0*.

definition *int* :: *nat* ⇒ *int* **where**
  [*code-abbrev*]: *int* = *of-nat*

code-const *int*
  (*SML* -)
  (*OCaml* -)

code-const *nat*

(*SML IntInf.max/ (0,/ -)*)
(*OCaml Big'-int.max'-big'-int/ Big'-int.zero'-big'-int*)
(*Eval Integer.max/ 0*)

For Haskell and Scala, things are slightly different again.

code-const *int* **and** *nat*
(*Haskell Prelude.toInteger* **and** *Prelude.fromInteger*)
(*Scala* !-.*as'-BigInt* **and** *Nat*)

Alternativ implementation for *of-nat*

lemma [*code*]:
of-nat n = (if n = 0 then 0 else
let
(q, m) = divmod-nat n 2;
q' = 2 ∗ of-nat q
in if m = 0 then q' else q' + 1)
proof −
from *mod-div-equality* **have** ∗: *of-nat n = of-nat (n div 2 ∗ 2 + n mod 2)* **by** *simp*
show *?thesis*
apply (*simp add*: *Let-def divmod-nat-div-mod mod-2-not-eq-zero-eq-one-nat of-nat-mult*
of-nat-add [*symmetric*])
apply (*auto simp add*: *of-nat-mult*)
apply (*simp add*: ∗ *of-nat-mult add-commute mult-commute*)
done
qed

Conversion from and to code numerals

code-const *Code-Numeral.of-nat*
(*SML IntInf.toInt*)
(*OCaml* -)
(*Haskell* !(*Prelude.fromInteger/ ./ Prelude.toInteger*))
(*Scala* !*Natural*(-.*as'-BigInt*))
(*Eval* -)

code-const *Code-Numeral.nat-of*
(*SML IntInf.fromInt*)
(*OCaml* -)
(*Haskell* !(*Prelude.fromInteger/ ./ Prelude.toInteger*))
(*Scala* !*Nat*(-.*as'-BigInt*))
(*Eval* -)

8.3 Target language arithmetic

code-const *plus* :: *nat ⇒ nat ⇒ nat*
(*SML IntInf.+/ ((-),/ (-))*)
(*OCaml Big'-int.add'-big'-int*)
(*Haskell* **infixl** *6* +)

(*Scala* **infixl** *7* +)
(*Eval* **infixl** *8* +)

code-const *minus* :: *nat* ⇒ *nat* ⇒ *nat*
(*SML IntInf.max/* (*0*, *IntInf.−/* ((-),/ (-))))
(*OCaml Big'-int.max'-big'-int/ Big'-int.zero'-big'-int/* (*Big'-int.sub'-big'-int/* -/ -))
(*Haskell* **infixl** *6* −)
(*Scala* **infixl** *7* −)
(*Eval Integer.max/ 0/* (- −/ -))

code-const *Code-Binary-Nat.dup*
(*SML IntInf.∗/* (*2*,/ (-)))
(*OCaml Big'-int.mult'-big'-int/ 2*)
(*Haskell* !(*2* ∗ -))
(*Scala* !(*2* ∗ -))
(*Eval* !(*2* ∗ -))

code-const *Code-Binary-Nat.sub*
(*SML* !(*raise/ Fail/ sub*))
(*OCaml failwith/ sub*)
(*Haskell error/ sub*)
(*Scala* !*sys.error*(*sub*))

code-const *times* :: *nat* ⇒ *nat* ⇒ *nat*
(*SML IntInf.∗/* ((-),/ (-)))
(*OCaml Big'-int.mult'-big'-int*)
(*Haskell* **infixl** *7* ∗)
(*Scala* **infixl** *8* ∗)
(*Eval* **infixl** *9* ∗)

code-const *divmod-nat*
(*SML IntInf.divMod/* ((-),/ (-)))
(*OCaml Big'-int.quomod'-big'-int*)
(*Haskell divMod*)
(*Scala* **infixl** *8* /%)
(*Eval Integer.div'-mod*)

code-const *HOL.equal* :: *nat* ⇒ *nat* ⇒ *bool*
(*SML* !((- : *IntInf.int*) = -))
(*OCaml Big'-int.eq'-big'-int*)
(*Haskell* **infix** *4* ==)
(*Scala* **infixl** *5* ==)
(*Eval* **infixl** *6* =)

code-const *less-eq* :: *nat* ⇒ *nat* ⇒ *bool*
(*SML IntInf.<=/* ((-),/ (-)))
(*OCaml Big'-int.le'-big'-int*)
(*Haskell* **infix** *4* <=)

(*Scala* **infixl** *4* <=)
(*Eval* **infixl** *6* <=)

code-const *less* :: *nat* ⇒ *nat* ⇒ *bool*
(*SML IntInf*.</ ((-),/ (-)))
(*OCaml Big′-int.lt′-big′-int*)
(*Haskell* **infix** *4* <)
(*Scala* **infixl** *4* <)
(*Eval* **infixl** *6* <)

code-const *Num.num-of-nat*
(*SML* !(*raise*/ *Fail*/ *num′-of′-nat*))
(*OCaml failwith*/ *num′-of′-nat*)
(*Haskell error*/ *num′-of′-nat*)
(*Scala* !*sys*.*error*(*num′-of′-nat*))

8.4 Evaluation

lemma [*code*, *code del*]:
(*Code-Evaluation.term-of* :: *nat* ⇒ *term*) = *Code-Evaluation.term-of* **..**

code-const *Code-Evaluation.term-of* :: *nat* ⇒ *term*
(*SML HOLogic.mk′-number*/ *HOLogic.natT*)

Evaluation with *Quickcheck-Narrowing* does not work yet, since a couple of questions how to perform evaluations in Haskell are not that clear yet. Therefore, we simply deactivate the narrowing-based quickcheck from here on.

declare [[*quickcheck-narrowing-active* = *false*]]

code-modulename *SML*
Efficient-Nat Arith

code-modulename *OCaml*
Efficient-Nat Arith

code-modulename *Haskell*
Efficient-Nat Arith

hide-const (**open**) *int*

end

9 Immutable Arrays with Code Generation

theory *IArray*

imports ~~/src/HOL/Library/Efficient-Nat
begin

Immutable arrays are lists wrapped up in an additional constructor. There are no update operations. Hence code generation can safely implement this type by efficient target language arrays. Currently only SML is provided. Should be extended to other target languages and more operations.

Note that arrays cannot be printed directly but only by turning them into lists first. Arrays could be converted back into lists for printing if they were wrapped up in an additional constructor.

datatype *'a iarray = IArray 'a list*

fun *of-fun* :: (*nat* ⇒ *'a*) ⇒ *nat* ⇒ *'a iarray* **where**
of-fun f n = IArray (map f [0..<n])
hide-const (**open**) *of-fun*

fun *length* :: *'a iarray* ⇒ *nat* **where**
length (IArray xs) = List.length xs
hide-const (**open**) *length*

fun *sub* :: *'a iarray* ⇒ *nat* ⇒ *'a* (**infixl** !! *100*) **where**
(IArray as) !! n = as!n
hide-const (**open**) *sub*

fun *list-of* :: *'a iarray* ⇒ *'a list* **where**
list-of (IArray xs) = xs
hide-const (**open**) *list-of*

9.1 Code Generation

code-type *iarray*
 (*SML - Vector.vector*)

code-const *IArray*
 (*SML Vector.fromList*)
code-const *IArray.sub*
 (*SML Vector.sub*(-,-))
code-const *IArray.length*
 (*SML Vector.length*)
code-const *IArray.of-fun*
 (*SML* !(*fn f* => *fn n* => *Vector.tabulate*(*n*,*f*)))

lemma *list-of-code*[*code*]:
 IArray.list-of A = map (%n. A!!n) [0..< IArray.length A]
by (*cases A*) (*simp add*: *map-nth*)

end

theory *IArray-Addenda*
imports
$ISABELLE-HOME/src/HOL/Library/IArray
begin

lemma *conjI4*:
assumes *A B C D*
shows $A \wedge B \wedge C \wedge D$
using *assms* **by** *simp*

10 Some previous instances

instantiation *iarray* :: (*plus*) *plus*
begin
definition *plus-iarray* :: *'a iarray* ⇒ *'a iarray* ⇒ *'a iarray*
where *plus-iarray A B* = *IArray.of-fun* (λ*n*. *A*!!*n* + *B* !! *n*) (*IArray.length A*)
instance proof qed
end

11 Some previous definitions and properties for IArrays

11.1 Lemmas

lemma *IArray-equality*:
assumes *length-eq*: *length A* = *length B*
shows (*IArray A* = *IArray B*) = (∀ *i*<*length A*. *IArray A*!!*i*=*IArray B*!!*i*)
by (*metis length-eq nth-equalityI sub.simps*)

lemma *IArray-equality'*:
assumes *length-eq*: *length A* = *length B*
and (∀ *i*<*length A*. *IArray A*!!*i*=*IArray B*!!*i*)
shows (*IArray A* = *IArray B*)
using *IArray-equality assms* **by** *blast*

lemma *of-fun-nth*:
assumes *i*: *i*<*n*
shows (*IArray.of-fun f n*) !! *i* = *f i*
unfolding *IArray.of-fun.simps* **using** *map-nth i* **by** *auto*

11.2 Definitions

fun *find* :: (*'a* => *bool*) => *'a iarray* => *'a option*
where *find f* (*IArray as*) = *List.find f as*
hide-const (**open**) *find*

fun *exists* :: (*'a* => *bool*) => *'a iarray* => *bool*
where *exists f* (*IArray as*) = (*if Option.is-none* (*IArray-Addenda.find f* (*IArray as*)) *then False else True*)
hide-const (**open**) *exists*

fun *all* :: (*'a* ⇒ *bool*) => *'a iarray* => *bool*
where *all f* (*IArray as*) = (*Option.is-none* (*IArray-Addenda.find* (λ*x*. ¬ *f x*) (*IArray as*)))
hide-const (**open**) *all*

primrec *findi-acc-list* :: (*nat* × *'a* => *bool*) => *'a list* => *nat* => (*nat* × *'a*) *option* **where**
findi-acc-list - [] *aux* = *None* |
findi-acc-list P (*x#xs*) *aux* = (*if P* (*aux*,*x*) *then Some* (*aux*,*x*) *else findi-acc-list P xs* (*Suc aux*))

definition *findi-list P x* = *findi-acc-list P x 0*

fun *findi* :: (*nat* × *'a* ⇒ *bool*) => *'a iarray* => (*nat* × *'a*) *option*
where *findi f* (*IArray as*) = *findi-list f as*
hide-const (**open**) *findi*

fun *foldl* :: ((*'a* × *'b*) ⇒ *'b*) => *'b* => *'a iarray* =>*'b*
where *foldl f a* (*IArray as*) = *List.foldl* (λ*x y*. *f* (*y*,*x*)) *a as*
hide-const (**open**) *foldl*

fun *filter* :: (*'a* ⇒ *bool*) => *'a iarray* => *'a iarray*
where *filter f* (*IArray as*) = *IArray* (*List.filter f as*)
hide-const (**open**) *filter*

11.3 Code generator

code-const *IArray-Addenda.find*
(*SML Vector.find*)

code-const *IArray-Addenda.exists*
(*SML Vector.exists*)

code-const *IArray-Addenda.all*
(*SML Vector.all*)

code-const *IArray-Addenda.findi*
(*SML Vector.findi*)

code-const *IArray-Addenda.foldl*
(*SML Vector.foldl*)

end

theory *Matrix-To-IArray*
 imports
 Gauss-Jordan-Examples
 IArray-Addenda
begin

12 Isomorphism between matrices implemented by vecs and matrices implemented by iarrays

12.1 Isomorphism between vec and iarray

definition *vec-to-iarray* :: $'a\hat{}'n$::{*mod-type*} ⇒ $'a$ *iarray*
 where *vec-to-iarray* A = *IArray.of-fun* (λi. A \$ (*from-nat* i)) ($CARD('n)$)

definition *iarray-to-vec* :: $'a$ *iarray* ⇒ $'a\hat{}'n$::{*mod-type*}
 where *iarray-to-vec* A = (χ i. A !! (*to-nat* i))

lemma *vec-to-iarray-nth*:
 fixes A::$'a\hat{}'n$::{*finite*, *mod-type*}
 assumes i: $i{<}CARD('n)$
 shows (*vec-to-iarray* A) !! i = A \$ (*from-nat* i)
 unfolding *vec-to-iarray-def* **using** *of-fun-nth*[*OF* i] **.**

lemma *vec-to-iarray-nth'*:
 fixes A::$'a\hat{}'n$::{*mod-type*}
 shows (*vec-to-iarray* A) !! (*to-nat* i) = A \$ i
proof −
 have *to-nat-less-card*: *to-nat* $i{<}CARD('n)$ **using** *bij-to-nat*[**where** *?'a*=$'n$] **unfolding** *bij-betw-def* **by** *fastforce*
 show *?thesis* **unfolding** *vec-to-iarray-def* **unfolding** *of-fun-nth*[*OF to-nat-less-card*] *from-nat-to-nat-id* **..**
qed

lemma *iarray-to-vec-nth*:
 shows (*iarray-to-vec* A) \$ i = A !! (*to-nat* i)
 unfolding *iarray-to-vec-def* **by** *simp*

lemma *vec-to-iarray-morph*:
 fixes A::$'a\hat{}'n$::{*mod-type*}
 shows (A = B) = (*vec-to-iarray* A = *vec-to-iarray* B)

by (*metis vec-eq-iff vec-to-iarray-nth'*)

lemma *inj-vec-to-iarray*:
shows *inj vec-to-iarray*
using *vec-to-iarray-morph* **unfolding** *inj-on-def* **by** *blast*

lemma *iarray-to-vec-vec-to-iarray*:
fixes *A*::*'a^'n*::{*mod-type*}
shows *iarray-to-vec* (*vec-to-iarray A*)=*A*
proof (*unfold vec-to-iarray-def iarray-to-vec-def*, *vector*, *auto*)
fix *i*::*'n*
have *to-nat i*<*CARD*(*'n*) **using** *bij-to-nat*[**where** *?'a*=*'n*] **unfolding** *bij-betw-def*
by *auto*
thus *map* (λ*i*. *A* $ *from-nat i*) [*0*..<*CARD*(*'n*)] ! *to-nat i* = *A* $ *i* **by** *simp*
qed

lemma *vec-to-iarray-iarray-to-vec*:
assumes *length-eq*: *IArray.length A* = *CARD*(*'n*::{*mod-type*})
shows *vec-to-iarray* (*iarray-to-vec A*::*'a^'n*::{*mod-type*}) = *A*
proof (*unfold vec-to-iarray-def iarray-to-vec-def*, *vector*, *auto*)
obtain *xs* **where** *xs*: *A* = *IArray xs* **by** (*metis list-of.cases*)
show *IArray* (*map* (λ*i*. *A* !! *to-nat* (*from-nat i*::*'n*)) [*0*..<*CARD*(*'n*)]) = *A*
proof (*unfold xs*, *rule IArray-equality'*, *auto*)
show *CARD*(*'n*) = *length xs* **using** *length-eq* **unfolding** *xs* **by** *simp*
fix *i* **assume** *i*: *i* < *CARD*(*'n*)
show *xs* ! *to-nat* (*from-nat i*::*'n*) = *xs* ! *i* **unfolding** *to-nat-from-nat-id*[*OF i*]
..
qed
qed

lemma *length-vec-to-iarray*:
fixes *xa*::*'a^'n*::{*mod-type*}
shows *IArray.length* (*vec-to-iarray xa*) = *CARD*(*'n*)
unfolding *vec-to-iarray-def* **by** *simp*

12.2 Isomorphism between matrix and nested iarrays

definition *matrix-to-iarray* :: *'a^'n*::{*mod-type*}*^'m*::{*mod-type*} => *'a iarray iarray*
where *matrix-to-iarray A* = *IArray* (*map* (*vec-to-iarray* ∘ (*op* $ *A*) ∘ (*from-nat*::*nat*=>*'m*)) [*0*..<*CARD*(*'m*)])

definition *iarray-to-matrix* :: *'a iarray iarray* ⇒ *'a^'n*::{*mod-type*}*^'m*::{*mod-type*}
where *iarray-to-matrix A* = (χ *i j*. *A* !! (*to-nat i*) !! (*to-nat j*))

lemma *matrix-to-iarray-morph*:
fixes *A*::*'a^'n*::{*mod-type*}*^'m*::{*mod-type*}
shows (*A* = *B*) = (*matrix-to-iarray A* = *matrix-to-iarray B*)
unfolding *matrix-to-iarray-def* **apply** *simp*

unfolding *forall-from-nat-rw*[*of* λx. *vec-to-iarray* (A \$ x) = *vec-to-iarray* (B \$ x)]
by (*metis from-nat-to-nat-id vec-eq-iff vec-to-iarray-morph*)

lemma *matrix-to-iarray-eq-of-fun*:
fixes A::'*a*^'*columns*::{*mod-type*}^'*rows*::{*mod-type*}
assumes *vec-eq-f*: $\forall$ *i*. *vec-to-iarray* (A \$ i) = f (*to-nat* i)
and *n-eq-length*: n=*IArray.length* (*matrix-to-iarray* A)
shows *matrix-to-iarray* A = *IArray.of-fun* f n
unfolding *of-fun.simps matrix-to-iarray-def* **proof** (*rule IArray-equality'*, *auto*)
show ∗: *CARD*('*rows*) = n **using** *n-eq-length* **unfolding** *matrix-to-iarray-def* **by** *auto*
fix i **assume** i: i < *CARD*('*rows*)
hence *i-less-n*: i<n **using** ∗ i **by** *simp*
show *vec-to-iarray* (A \$ *mod-type-class.from-nat* i) = *map* f [*0*..<n] ! i
using *vec-eq-f* **using** *i-less-n*
by (*simp*, *unfold to-nat-from-nat-id*[*OF* i], *simp*)
qed

lemma *map-vec-to-iarray-rw*[*simp*]:
fixes A::'*a*^'*columns*::{*mod-type*}^'*rows*::{*mod-type*}
shows *map* (λx. *vec-to-iarray* (A \$ *from-nat* x)) [*0*..<*CARD*('*rows*)] ! *to-nat* i = *vec-to-iarray* (A \$ i)
proof −
have *i-less-card*: *to-nat* i < *CARD*('*rows*) **using** *bij-to-nat*[**where** *?'a*='*rows*] **unfolding** *bij-betw-def* **by** *fastforce*
hence *map* (λx. *vec-to-iarray* (A \$ *from-nat* x)) [*0*..<*CARD*('*rows*)] ! *to-nat* i = *vec-to-iarray* (A \$ *from-nat* (*to-nat* i)) **by** *simp*
also have ... = *vec-to-iarray* (A \$ i) **unfolding** *from-nat-to-nat-id* **..**
finally show *?thesis* **.**
qed

lemma *matrix-to-iarray-nth*:
matrix-to-iarray A !! *to-nat* i !! *to-nat* j = A \$ i \$ j
unfolding *matrix-to-iarray-def o-def* **using** *vec-to-iarray-nth'* **by** *auto*

lemma *vec-matrix*: *vec-to-iarray* (A\$$i$) = (*matrix-to-iarray* A) !! (*to-nat* i)
unfolding *matrix-to-iarray-def o-def* **by** *fastforce*

lemma *iarray-to-matrix-matrix-to-iarray*:
fixes A::'*a*^'*columns*::{*mod-type*}^'*rows*::{*mod-type*}
shows *iarray-to-matrix* (*matrix-to-iarray* A)=A **unfolding** *matrix-to-iarray-def iarray-to-matrix-def o-def* **apply** *vector* **apply** *auto*
unfolding *vec-to-iarray-nth'* **..**

13 Definition of operations over matrices implemented by iarrays

definition *mult-iarray* :: 'a::{times} iarray => 'a => 'a iarray
where *mult-iarray A q = IArray.of-fun (λn. q * A!!n) (IArray.length A)*

definition *row-iarray* :: nat => 'a iarray iarray => 'a iarray
where *row-iarray k A = A !! k*

definition *column-iarray* :: nat => 'a iarray iarray => 'a iarray
where *column-iarray k A = IArray.of-fun (λm. A !! m !! k) (IArray.length A)*

definition *nrows-iarray* :: 'a iarray iarray => nat
where *nrows-iarray A = IArray.length A*

definition *ncols-iarray* :: 'a iarray iarray => nat
where *ncols-iarray A = IArray.length (A!!0)*

definition *rows-iarray A = {row-iarray i A | i. i ∈ {0..<nrows-iarray A}}*
definition *columns-iarray A = {column-iarray i A | i. i ∈ {0..<ncols-iarray A}}*

definition *tabulate2* :: nat => nat => (nat => nat => 'a) => 'a iarray iarray
where *tabulate2 m n f = IArray.of-fun (λi. IArray.of-fun (f i) n) m*

definition *transpose-iarray* :: 'a iarray iarray => 'a iarray iarray
where *transpose-iarray A = tabulate2 (ncols-iarray A) (nrows-iarray A) (λa b. A!!b!!a)*

definition *matrix-matrix-mult-iarray* :: 'a::{times, comm-monoid-add} iarray iarray => 'a iarray iarray => 'a iarray iarray (**infixl** **i 70)
where *A **i B = tabulate2 (nrows-iarray A) (ncols-iarray B) (λi j. setsum (λk. ((A!!i)!!k) * ((B!!k)!!j)) {0..<ncols-iarray A})*

definition *matrix-vector-mult-iarray* :: 'a::{semiring-1} iarray iarray => 'a iarray => 'a iarray (**infixl** *iv 70)
where *A *iv x = IArray.of-fun (λi. setsum (λj. ((A!!i)!!j) * (x!!j)) {0..<IArray.length x}) (nrows-iarray A)*

definition *vector-matrix-mult-iarray* :: 'a::{semiring-1} iarray => 'a iarray iarray => 'a iarray (**infixl** v*i 70)
where *x v*i A = IArray.of-fun (λj. setsum (λi. ((A!!i)!!j) * (x!!i)) {0..<IArray.length x}) (ncols-iarray A)*

definition *mat-iarray* :: 'a::{zero} => nat => 'a iarray iarray
where *mat-iarray k n = tabulate2 n n (λ i j. if i = j then k else 0)*

definition *is-zero-iarray* :: 'a::{zero} iarray ⇒ bool
where *is-zero-iarray A = IArray-Addenda.all (λi. A !! i = 0) (IArray[0..<IArray.length*

A])

13.1 Properties of previous definitions

lemma *is-zero-iarray-eq-iff*:
fixes *A*::'*a*::{*zero*} ^'*n*::{*mod-type*}
shows (*A* = *0*) = (*is-zero-iarray* (*vec-to-iarray A*))
proof (*auto*)
show *is-zero-iarray* (*vec-to-iarray 0*) **by** (*simp add*: *vec-to-iarray-def is-zero-iarray-def is-none-def find-None-iff*)
show *is-zero-iarray* (*vec-to-iarray A*) ⟹ *A* = *0*
proof (*simp add*: *vec-to-iarray-def is-zero-iarray-def is-none-def find-None-iff vec-eq-iff*, *clarify*)
fix *i*::'*n*
assume *eq-zero*: ∀ *x*<*CARD*('*n*). *A* $ *from-nat x* = *0*
have *to-nat i*<*CARD*('*n*) **using** *bij-to-nat*[**where** *?'a*='*n*] **unfolding** *bij-betw-def* **by** *fastforce*
hence *A* $ (*from-nat* (*to-nat i*)) = *0* **using** *eq-zero* **by** *blast*
thus *A* $ *i* = *0* **unfolding** *from-nat-to-nat-id* **.**
qed
qed

lemma *mult-iarray-works*:
assumes *a*<*IArray.length A* **shows** *mult-iarray A q* !! *a* = *q* ∗ *A*!!*a*
unfolding *mult-iarray-def* **unfolding** *of-fun.simps* **unfolding** *sub.simps*
using *assms* **by** *simp*

lemma *length-eq-card-rows*:
fixes *A*::'*a* ^'*columns*::{*mod-type*} ^'*rows*::{*mod-type*}
shows *IArray.length* (*matrix-to-iarray A*) = *CARD*('*rows*)
unfolding *matrix-to-iarray-def* **by** *auto*

lemma *nrows-eq-card-rows*:
fixes *A*::'*a* ^'*columns*::{*mod-type*} ^'*rows*::{*mod-type*}
shows *nrows-iarray* (*matrix-to-iarray A*) = *CARD*('*rows*)
unfolding *nrows-iarray-def length-eq-card-rows* **..**

lemma *length-eq-card-columns*:
fixes *A*::'*a* ^'*columns*::{*mod-type*} ^'*rows*::{*mod-type*}
shows *IArray.length* (*matrix-to-iarray A* !! *0*) = *CARD* ('*columns*)
unfolding *matrix-to-iarray-def o-def vec-to-iarray-def* **by** *simp*

lemma *ncols-eq-card-columns*:
fixes *A*::'*a* ^'*columns*::{*mod-type*} ^'*rows*::{*mod-type*}
shows *ncols-iarray* (*matrix-to-iarray A*) = *CARD*('*columns*)
unfolding *ncols-iarray-def length-eq-card-columns* **..**

lemma *matrix-to-iarray-nrows*:
fixes *A*::'*a* ^'*columns*::{*mod-type*} ^'*rows*::{*mod-type*}

shows *nrows A* = *nrows-iarray* (*matrix-to-iarray A*)
unfolding *nrows-def nrows-eq-card-rows* **..**

lemma *matrix-to-iarray-ncols*:
fixes *A*::′*a*^′*columns*::{*mod-type*}^′*rows*::{*mod-type*}
shows *ncols A* = *ncols-iarray* (*matrix-to-iarray A*)
unfolding *ncols-def ncols-eq-card-columns* **..**

lemma *vec-to-iarray-row*[*code-unfold*]: *vec-to-iarray* (*row i A*) = *row-iarray* (*to-nat i*) (*matrix-to-iarray A*)
unfolding *row-def row-iarray-def vec-to-iarray-def*
by (*auto*, *metis of-fun.simps vec-matrix vec-to-iarray-def*)

lemma *vec-to-iarray-row*′: *vec-to-iarray* (*row i A*) = (*matrix-to-iarray A*) !! (*to-nat i*)
unfolding *row-def vec-to-iarray-def*
by (*auto*, *metis of-fun.simps vec-matrix vec-to-iarray-def*)

lemma *vec-to-iarray-column*[*code-unfold*]: *vec-to-iarray* (*column i A*) = *column-iarray* (*to-nat i*) (*matrix-to-iarray A*)
unfolding *column-def vec-to-iarray-def column-iarray-def length-eq-card-rows*
by (*auto*, *metis from-nat-not-eq vec-matrix vec-to-iarray-nth*′)

lemma *vec-to-iarray-rows*: *vec-to-iarray*‘ (*rows A*) = *rows-iarray* (*matrix-to-iarray A*)
unfolding *rows-def* **unfolding** *rows-iarray-def*
apply (*auto simp add*: *vec-to-iarray-row to-nat-less-card nrows-eq-card-rows*)
by (*unfold image-def*, *auto*, *metis from-nat-not-eq vec-to-iarray-row*)

lemma *vec-to-iarray-columns*: *vec-to-iarray*‘ (*columns A*) = *columns-iarray* (*matrix-to-iarray A*)
unfolding *columns-def* **unfolding** *columns-iarray-def*
apply(*auto simp add*: *ncols-eq-card-columns to-nat-less-card vec-to-iarray-column*)
by (*unfold image-def*, *auto*, *metis from-nat-not-eq vec-to-iarray-column*)

14 Definition of elementary operations

definition *interchange-rows-iarray* :: ′*a iarray iarray* => *nat* => *nat* => ′*a iarray iarray*
where *interchange-rows-iarray A a b* = *IArray.of-fun* (λ*n. if n*=*a then A*!!*b else if n*=*b then A*!!*a else A*!!*n*) (*IArray.length A*)

definition *mult-row-iarray* :: ′*a*::{*times*} *iarray iarray* => *nat* => ′*a* => ′*a iarray iarray*
where *mult-row-iarray A a q* = *IArray.of-fun* (λ*n. if n*=*a then mult-iarray* (*A*!!*a*) *q else A*!!*n*) (*IArray.length A*)

definition *row-add-iarray* :: ′*a*::{*plus*, *times*} *iarray iarray* => *nat* => *nat* => ′*a* => ′*a iarray iarray*

where *row-add-iarray A a b q* = *IArray.of-fun* (λ*n. if n=a then A!!a* + *mult-iarray* (*A!!b*) *q else A!!n*) (*IArray.length A*)

definition *interchange-columns-iarray* :: *'a iarray iarray => nat => nat => 'a iarray iarray*
where *interchange-columns-iarray A a b* = *tabulate2* (*nrows-iarray A*) (*ncols-iarray A*) (λ*i j. if j* = *a then A !! i !! b else if j* = *b then A !! i !! a else A !! i !! j*)

definition *mult-column-iarray* :: *'a::{times} iarray iarray => nat => 'a => 'a iarray iarray*
where *mult-column-iarray A n q* = *tabulate2* (*nrows-iarray A*) (*ncols-iarray A*) (λ*i j. if j* = *n then A !! i !! j* ∗ *q else A !! i !! j*)

definition *column-add-iarray* :: *'a::{plus, times} iarray iarray => nat => nat => 'a => 'a iarray iarray*
where *column-add-iarray A n m q* = *tabulate2* (*nrows-iarray A*) (*ncols-iarray A*) (λ*i j. if j* = *n then A !! i !! n* + *A !! i !! m* ∗ *q else A !! i !! j*)

14.1 Code generator

lemma *vec-to-iarray-plus*[*code-unfold*]: *vec-to-iarray* (*a* + *b*) = (*vec-to-iarray a*) + (*vec-to-iarray b*)
unfolding *vec-to-iarray-def*
unfolding *plus-iarray-def* **by** *auto*

lemma *matrix-to-iarray-plus*[*code-unfold*]: *matrix-to-iarray* (*A* + *B*) = (*matrix-to-iarray A*) + (*matrix-to-iarray B*)
unfolding *matrix-to-iarray-def o-def*
by (*simp add*: *plus-iarray-def vec-to-iarray-plus*)

lemma *matrix-to-iarray-mat*[*code-unfold*]:
matrix-to-iarray (*mat k* :: *'a::{zero} ^'n::{mod-type} ^'n::{mod-type}*) = *mat-iarray k CARD*(*'n::{mod-type}*)
unfolding *matrix-to-iarray-def o-def vec-to-iarray-def mat-def mat-iarray-def tabulate2-def*
using *from-nat-eq-imp-eq* **by** *fastforce*

lemma *matrix-to-iarray-transpose*[*code-unfold*]:
shows *matrix-to-iarray* (*transpose A*) = *transpose-iarray* (*matrix-to-iarray A*)
unfolding *matrix-to-iarray-def transpose-def transpose-iarray-def*
o-def tabulate2-def nrows-iarray-def ncols-iarray-def vec-to-iarray-def
by *auto*

lemma *matrix-to-iarray-matrix-matrix-mult*[*code-unfold*]:
fixes *A::'a::{semiring-1} ^'m::{mod-type} ^'n::{mod-type}* **and** *B::'a ^'b::{mod-type} ^'m::{mod-type}*
shows *matrix-to-iarray* (*A* ∗∗ *B*) = (*matrix-to-iarray A*) ∗∗*i* (*matrix-to-iarray B*)
unfolding *matrix-to-iarray-def matrix-matrix-mult-iarray-def matrix-matrix-mult-def*

unfolding *o-def tabulate2-def nrows-iarray-def ncols-iarray-def vec-to-iarray-def*
using *setsum-reindex-cong*[*of from-nat*::*nat*=>′*m*] **using** *bij-from-nat* **unfolding** *bij-betw-def* **by** *fastforce*

lemma *vec-to-iarray-matrix-matrix-mult*[*code-unfold*]:
fixes *A*::′*a*::{*semiring-1*} ^′*m*::{*mod-type*} ^′*n*::{*mod-type*} **and** *x*::′*a* ^′*m*::{*mod-type*}
shows *vec-to-iarray* (*A* ∗*v* *x*) = (*matrix-to-iarray* *A*) ∗*iv* (*vec-to-iarray* *x*)
unfolding *matrix-vector-mult-iarray-def matrix-vector-mult-def*
unfolding *o-def tabulate2-def nrows-iarray-def ncols-iarray-def matrix-to-iarray-def vec-to-iarray-def*
using *setsum-reindex-cong*[*of from-nat*::*nat*=>′*m*] **using** *bij-from-nat* **unfolding** *bij-betw-def* **by** *fastforce*

lemma *vec-to-iarray-vector-matrix-mult*[*code-unfold*]:
fixes *A*::′*a*::{*semiring-1*} ^′*m*::{*mod-type*} ^′*n*::{*mod-type*} **and** *x*::′*a* ^′*n*::{*mod-type*}
shows *vec-to-iarray* (*x* *v*∗ *A*) = (*vec-to-iarray* *x*) *v*∗*i* (*matrix-to-iarray* *A*)
unfolding *vector-matrix-mult-def vector-matrix-mult-iarray-def*
unfolding *o-def tabulate2-def nrows-iarray-def ncols-iarray-def matrix-to-iarray-def vec-to-iarray-def*
proof (*auto*)
fix *xa*
show ($\sum$ *i*∈*UNIV*. *A* \$ *i* \$ *from-nat xa* ∗ *x* \$ *i*) = ($\sum$ *i* = *0*..<*CARD*(′*n*). *A* \$ *from-nat i* \$ *from-nat xa* ∗ *x* \$ *from-nat i*)
apply (*rule setsum-reindex-cong*[*of from-nat*::*nat*=>′*n*]) **using** *bij-from-nat*[**where** *?′a*=′*n*] **unfolding** *bij-betw-def* **by** *fast*+
qed

lemma *matrix-to-iarray-interchange-rows*[*code-unfold*]:
fixes *A*::′*a*::{*semiring-1*} ^′*columns*::{*mod-type*} ^′*rows*::{*mod-type*}
shows *matrix-to-iarray* (*interchange-rows A i j*) = *interchange-rows-iarray* (*matrix-to-iarray A*) (*to-nat i*) (*to-nat j*)
unfolding *matrix-to-iarray-def interchange-rows-iarray-def o-def* **unfolding** *map-vec-to-iarray-rw*
apply *auto*
proof −
fix *x* **assume** *x-less-card*: *x* < *CARD*(′*rows*)
and *x-not-j*: *x* ≠ *to-nat j* **and** *x-not-i*: *x* ≠ *to-nat i*
show *vec-to-iarray* (*interchange-rows A i j* \$ *from-nat x*) = *vec-to-iarray* (*A* \$ *from-nat x*)
by (*metis interchange-rows-preserves to-nat-from-nat-id x-less-card x-not-i x-not-j*)
qed

lemma *matrix-to-iarray-mult-row*[*code-unfold*]:
fixes *A*::′*a*::{*semiring-1*} ^′*columns*::{*mod-type*} ^′*rows*::{*mod-type*}
shows *matrix-to-iarray* (*mult-row A i q*) = *mult-row-iarray* (*matrix-to-iarray A*) (*to-nat i*) *q*

unfolding *matrix-to-iarray-def mult-row-iarray-def o-def*
unfolding *mult-iarray-def vec-to-iarray-def mult-row-def* **apply** *auto*
proof −
fix *i x*
assume *i-contr*:$i \neq$ *to-nat* (*from-nat* i::$'rows$) **and** $x <$ *CARD*($'columns$)
and $i <$*CARD*($'rows$)
hence $i =$ *to-nat* (*from-nat* i::$'rows$) **using** *to-nat-from-nat-id* **by** *fastforce*
thus $q * A$ \$ *mod-type-class.from-nat* i \$ *mod-type-class.from-nat* $x = A$ \$ *mod-type-class.from-nat* i \$ *mod-type-class.from-nat* x
using *i-contr* **by** *contradiction*
qed

lemma *matrix-to-iarray-row-add*[*code-unfold*]:
fixes A::$'a$::{*semiring-1*} ^$'columns$::{*mod-type*} ^$'rows$::{*mod-type*}
shows *matrix-to-iarray* (*row-add A i j q*) = *row-add-iarray* (*matrix-to-iarray A*) (*to-nat i*) (*to-nat j*) *q*
unfolding *matrix-to-iarray-def row-add-iarray-def o-def*
proof (*auto*)
show *vec-to-iarray* (*row-add A i j q* \$ *i*) = *vec-to-iarray* (*A* \$ *i*) + *mult-iarray* (*vec-to-iarray* (*A* \$ *j*)) *q*
unfolding *mult-iarray-def vec-to-iarray-def* **unfolding** *plus-iarray-def row-add-def* **by** *auto*
fix *ia* **assume** *ia-not-i*: $ia \neq$ *to-nat i* **and** *ia-card*: $ia <$ *CARD*($'rows$)
have *from-nat-ia-not-i*: *from-nat ia* $\neq i$
proof (*rule ccontr*)
assume $\neg$ *mod-type-class.from-nat ia* $\neq i$ **hence** *from-nat ia* $= i$ **by** *simp*
hence *to-nat* (*from-nat ia*::$'rows$) = *to-nat i* **by** *simp*
hence *ia*=*to-nat i* **using** *to-nat-from-nat-id ia-card* **by** *fastforce*
thus *False* **using** *ia-not-i* **by** *contradiction*
qed
show *vec-to-iarray* (*row-add A i j q* \$ *from-nat ia*) = *vec-to-iarray* (*A* \$ *from-nat ia*)
using *ia-not-i*
unfolding *vec-to-iarray-morph*[*symmetric*] **unfolding** *row-add-def* **using** *from-nat-ia-not-i* **by** *vector*
qed

lemma *matrix-to-iarray-interchange-columns*[*code-unfold*]:
fixes A::$'a$::{*semiring-1*} ^$'columns$::{*mod-type*} ^$'rows$::{*mod-type*}
shows *matrix-to-iarray* (*interchange-columns A i j*) = *interchange-columns-iarray* (*matrix-to-iarray A*) (*to-nat i*) (*to-nat j*)
unfolding *interchange-columns-def interchange-columns-iarray-def o-def tabulate2-def*
unfolding *nrows-eq-card-rows ncols-eq-card-columns*
unfolding *matrix-to-iarray-def o-def vec-to-iarray-def*
by (*auto simp add*: *to-nat-from-nat-id to-nat-less-card*[*of i*] *to-nat-less-card*[*of j*])

lemma *matrix-to-iarray-mult-columns*[*code-unfold*]:
fixes A::$'a$::{*semiring-1*} ^$'columns$::{*mod-type*} ^$'rows$::{*mod-type*}

shows *matrix-to-iarray* (*mult-column A i q*) = *mult-column-iarray* (*matrix-to-iarray A*) (*to-nat i*) *q*
unfolding *mult-column-def mult-column-iarray-def o-def tabulate2-def*
unfolding *nrows-eq-card-rows ncols-eq-card-columns*
unfolding *matrix-to-iarray-def o-def vec-to-iarray-def*
by (*auto simp add*: *to-nat-from-nat-id*)

lemma *matrix-to-iarray-column-add*[*code-unfold*]:
fixes *A*::′*a*::{*semiring-1*} ^′*columns*::{*mod-type*} ^′*rows*::{*mod-type*}
shows *matrix-to-iarray* (*column-add A i j q*) = *column-add-iarray* (*matrix-to-iarray A*) (*to-nat i*) (*to-nat j*) *q*
unfolding *column-add-def column-add-iarray-def o-def tabulate2-def*
unfolding *nrows-eq-card-rows ncols-eq-card-columns*
unfolding *matrix-to-iarray-def o-def vec-to-iarray-def*
by (*auto simp add*: *to-nat-from-nat-id to-nat-less-card*[*of i*] *to-nat-less-card*[*of j*])

14.2 Definitions and equivalences of null space, column space and row space.

definition *null-space-iarray* :: ′*a*::{*zero*, *semiring-1*} *iarray iarray* => ′*a iarray set*
where *null-space-iarray A* = {*x*. *is-zero-iarray* (*A* ∗*iv x*) ∧ *IArray.length x* = *ncols-iarray A*}

definition *col-space-iarray A* = {*y*. ∃ *x*. *A* ∗*iv x* = *y* ∧ *IArray.length x* = *ncols-iarray A*}

definition *row-space-iarray A* = *col-space-iarray* (*transpose-iarray A*)

The row space of a matrix represented as a nested iarrays can't be defined as follows due to IArray is not an instance of real vector

definition row-space-iarray A = *span* (*rows-iarray A*)

lemma *matrix-to-iarray-null-space*:
fixes *A*::*real*^′*columns*::{*mod-type*} ^′*rows*::{*mod-type*}
shows *vec-to-iarray*‘ (*null-space A*) = *null-space-iarray* (*matrix-to-iarray A*)
proof (*unfold image-def null-space-iarray-def null-space-def*, *auto*)
fix *x*
assume *x*: *is-zero-iarray* (*matrix-to-iarray A* ∗*iv x*)
and *length-x*: *IArray.length x* = *ncols-iarray* (*matrix-to-iarray A*)
show ∃ *xa*. *A* ∗*v xa* = *0* ∧ *x* = *vec-to-iarray xa*
proof (*rule exI*[*of - iarray-to-vec x*], *rule conjI*)
show *x* = *vec-to-iarray* (*iarray-to-vec x*::*real*^′*columns*::{*mod-type*})
by (*rule vec-to-iarray-iarray-to-vec*[*symmetric*], *simp add*: *length-x ncols-eq-card-columns*)
thus *A* ∗*v iarray-to-vec x* = *0* **by** (*metis is-zero-iarray-eq-iff vec-to-iarray-matrix-matrix-mult x*)
qed
next
fix *xa*::*real*^′*columns*::{*mod-type*}

show *IArray.length* (*vec-to-iarray xa*) = *ncols-iarray* (*matrix-to-iarray A*)
unfolding *length-vec-to-iarray ncols-eq-card-columns* **..**
assume *xa*: *A* ∗*v xa* = *0*
show *is-zero-iarray* (*matrix-to-iarray A* ∗*iv vec-to-iarray xa*)
unfolding *vec-to-iarray-matrix-matrix-mult*[*symmetric*] *xa is-zero-iarray-eq-iff* [*symmetric*]
..
qed

lemma *matrix-to-iarray-col-space*:
fixes *A*::*real*^′*columns*::{*mod-type*} ^′*rows*::{*mod-type*}
shows *vec-to-iarray*‘ (*col-space A*) = *col-space-iarray* (*matrix-to-iarray A*)
unfolding *col-space-eq* **unfolding** *Dim-Formula.col-space-eq*
unfolding *image-def col-space-iarray-def*
apply (*auto*, *metis length-vec-to-iarray ncols-eq-card-columns vec-to-iarray-matrix-matrix-mult*)
by (*metis ncols-eq-card-columns vec-to-iarray-iarray-to-vec vec-to-iarray-matrix-matrix-mult*)

lemma *matrix-to-iarray-row-space*:
fixes *A*::*real*^′*columns*::{*mod-type*} ^′*rows*::{*mod-type*}
shows *vec-to-iarray*‘ (*row-space A*) = *row-space-iarray* (*matrix-to-iarray A*)
unfolding *row-space-iarray-def row-space-def*
unfolding *matrix-to-iarray-transpose*[*symmetric*] *matrix-to-iarray-col-space*[*symmetric*]
unfolding *col-space-def columns-transpose* **..**

end

theory *Gauss-Jordan-IArrays*
imports *Matrix-To-IArray*
begin

14.3 Definitions and functions to compute the Gauss-Jordan algorithm over matrices represented as nested iarrays

definition *least-n* :: ′*a*::{*zero*} *iarray iarray* => *nat* => *nat* => *nat*
where *least-n A i j* = (*the* (*List.find* (λ*x*. *A* !! *x* !! *j* ≠ *0*) [*i*..<*nrows-iarray A*]))

definition *all-zero-from-k* :: (*nat* × ′*a*::{*field*} *iarray iarray*) => *nat* => *bool*
where *all-zero-from-k A′ k* = (*let i*=*fst A′*; *A*=(*snd A′*) *in IArray-Addenda.all* (λ*x*. *A*!!*x*!!*k* = *0*) (*IArray* [*i*..<(*nrows-iarray A*)]))

definition *Gauss-Jordan-in-ij-iarrays* :: ′*a*::{*field*} *iarray iarray* => *nat* => *nat* => ′*a iarray iarray*
where *Gauss-Jordan-in-ij-iarrays A i j* = (*let n* = *least-n A i j*;
interchange-A = *interchange-rows-iarray A i n*;
A′ = *mult-row-iarray interchange-A i* (*1*/*interchange-A*!!*i*!!*j*)
in IArray.of-fun (λ*s*. *if s* = *i then A′* !! *s else row-add-iarray A′ s i* (− *interchange-A* !! *s* !! *j*) !! *s*) (*nrows-iarray A*))

definition *Gauss-Jordan-column-k-iarrays* :: (*nat* × *'a*::{*field*} *iarray iarray*) => *nat* => (*nat* × *'a iarray iarray*)
where *Gauss-Jordan-column-k-iarrays A' k*=(*let A*=(*snd A'*); *i*=(*fst A'*) *in*
if (*all-zero-from-k A' k*) ∨ *i* = (*nrows-iarray A*) *then* (*i*,*A*) *else* (*Suc i*, (*Gauss-Jordan-in-ij-iarrays A i k*)))

definition *Gauss-Jordan-upt-k-iarrays* :: *'a*::{*field*} *iarray iarray* => *nat* => *'a*::{*field*} *iarray iarray*
where *Gauss-Jordan-upt-k-iarrays A k* = *snd* (*foldl Gauss-Jordan-column-k-iarrays* (*0*,*A*) [*0*..<*Suc k*])

definition *Gauss-Jordan-iarrays* :: *'a*::{*field*} *iarray iarray* => *'a*::{*field*} *iarray iarray*
where *Gauss-Jordan-iarrays A* = *Gauss-Jordan-upt-k-iarrays A* (*ncols-iarray A* − *1*)

definition *rank-iarray* :: *'a*::{*field*} *iarray iarray* => *nat*
where *rank-iarray A* = (*let A'* = (*Gauss-Jordan-iarrays A*); *nrows* = (*IArray.length A'*) *in card* {*i*. *i*<*nrows* ∧ ¬ *is-zero-iarray* (*A'* !! *i*)})

14.4 Proving the equivalence between Gauss-Jordan algorithm over nested iarrays and over nested vecs.

lemma *all-zero-from-k-eq*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*}
shows (∀ *m*≥*i*. *A* $ *m* $ *k* = *0*) = (*all-zero-from-k* (*to-nat i*, *matrix-to-iarray A*) (*to-nat k*))
proof (*auto simp add*: *all-zero-from-k-def Let-def is-none-def find-None-iff*)
fix *x*
assume *zero*: ∀ *m*≥*i*. *A* $ *m* $ *k* = *0*
and *x-length*: *x* < *nrows-iarray* (*matrix-to-iarray A*) **and** *i-le-x*: *to-nat i* ≤ *x*
have *x-le-card*: *x* < *CARD*(*'rows*) **using** *length-eq-card-rows x-length* **unfolding** *nrows-iarray-def* **by** *metis*
have *i-le-from-nat-x*: *i* ≤ *from-nat x* **using** *from-nat-mono'*[*OF i-le-x x-le-card*] **unfolding** *from-nat-to-nat-id* **.**
hence *Axk*: *A* $ (*from-nat x*) $ *k* = *0* **using** *zero* **by** *simp*
have *matrix-to-iarray A* !! *x* !! *to-nat k* = *matrix-to-iarray A* !! *to-nat* (*from-nat x*::*'rows*) !! *to-nat k* **unfolding** *to-nat-from-nat-id*[*OF x-le-card*] **..**
also have ... = *A* $ (*from-nat x*) $ *k* **unfolding** *matrix-to-iarray-nth* **..**
also have ... = *0* **unfolding** *Axk* **..**
finally show *matrix-to-iarray A* !! *x* !! *to-nat k* = *0* **.**
next
fix *m*::*'rows*
assume *zero*: ∀ *x*< *nrows-iarray* (*matrix-to-iarray A*). *to-nat i* ≤ *x* ⟶ *matrix-to-iarray A* !! *x* !! *to-nat k* = *0* **and** *i-le-m*: *i* ≤ *m*
have *to-nat-i-le-m*:*to-nat i* ≤ *to-nat m* **using** *to-nat-mono'*[*OF i-le-m*] **.**
have *m-le-length*: *to-nat m* < *IArray.length* (*matrix-to-iarray A*) **unfolding** *length-eq-card-rows* **using** *bij-to-nat*[**where** *?'a*=*'rows*] **unfolding** *bij-betw-def* **by** *force*

have A $ m $ k = matrix-to-iarray A !! (to-nat m) !! to-nat k **unfolding** matrix-to-iarray-nth **..**
also have ... = 0 **using** zero to-nat-i-le-m m-le-length **unfolding** nrows-iarray-def **by** blast
finally show A $ m $ k = 0 **.**
qed

lemma matrix-to-iarray-least-n:
fixes A::'a::{field} ^'columns::{mod-type} ^'rows::{mod-type}
assumes not-all-zero: ¬ (all-zero-from-k (to-nat i, matrix-to-iarray A) (to-nat j))
shows least-n (matrix-to-iarray A) (to-nat i) (to-nat j) = to-nat (LEAST n. A $ n $ j ≠ 0 ∧ i ≤ n)
proof −
obtain a **where** a: List.find (λx. matrix-to-iarray A !! x !! (to-nat j) ≠ 0) [to-nat i..<nrows-iarray (matrix-to-iarray A)] = Some a
using not-all-zero **unfolding** all-zero-from-k-def Let-def **by** (auto simp add: is-none-def)
from this **obtain** ia **where**
ia-less-length: ia<length [to-nat i..<nrows-iarray (matrix-to-iarray A)] **and**
not-eq-zero: matrix-to-iarray A !! ([to-nat i..<nrows-iarray (matrix-to-iarray A)] ! ia) !! (to-nat j) ≠ 0 **and**
a-eq: a = [to-nat i..<nrows-iarray (matrix-to-iarray A)] ! ia
and least: (∀ja<ia. ¬ matrix-to-iarray A !! ([to-nat i..<nrows-iarray (matrix-to-iarray A)] ! ja) !! (to-nat j) ≠ 0)
unfolding find-Some-iff **by** blast
have not-eq-zero': matrix-to-iarray A !! a !! (to-nat j) ≠ 0 **using** not-eq-zero **unfolding** a-eq **.**
have i-less-a: to-nat i ≤ a **using** ia-less-length length-upt nth-upt a-eq **by** auto
have a-less-card: a<CARD('rows) **using** a-eq ia-less-length **unfolding** length-eq-card-rows nrows-iarray-def **by** auto
have (LEAST n. A $ n $ j ≠ 0 ∧ i ≤ n) = from-nat a
proof (rule Least-equality, rule conjI)
show A $ from-nat a $ j ≠ 0 **unfolding** matrix-to-iarray-nth[symmetric] **using** not-eq-zero' **unfolding** to-nat-from-nat-id[OF a-less-card] **.**
show i ≤ from-nat a **using** a-less-card from-nat-mono' from-nat-to-nat-id i-less-a **by** fastforce
fix x **assume** A $ x $ j ≠ 0 ∧ i ≤ x **hence** Axj: A $ x $ j ≠ 0 **and** i-le-x: i ≤ x **by** fast+
show from-nat a ≤ x
proof (rule ccontr) **thm** least
assume ¬ from-nat a ≤ x **hence** x-less-from-nat-a: x < from-nat a **by** simp
def ja≡(to-nat x) − (to-nat i)
have to-nat-x-less-card: to-nat x < CARD ('rows) **using** bij-to-nat[**where** ?'a='rows] **unfolding** bij-betw-def **by** fastforce
hence ja-less-length: ja < nrows-iarray (matrix-to-iarray A) **unfolding** ja-def length-eq-card-rows nrows-iarray-def **by** auto
have [to-nat i..<nrows-iarray (matrix-to-iarray A)] ! ja = to-nat i + ja

by (*rule nth-upt*, *auto simp add*: *ja-def to-nat-x-less-card length-eq-card-rows i-le-x to-nat-mono′ nrows-iarray-def*)
also have *i-plus-ja*: ... = *to-nat x* **unfolding** *ja-def* **by** (*simp add*: *i-le-x to-nat-mono′*)
finally have *list-rw*: [*to-nat i*..<*nrows-iarray* (*matrix-to-iarray A*)] ! *ja* = *to-nat x* **.**
moreover have *ja*<*ia*
proof −
have *a* = *to-nat i* + *ia* **unfolding** *a-eq* **by** (*rule nth-upt*, *metis ia-less-length length-upt less-diff-conv nat-add-commute*)
thus *?thesis* **by** (*metis i-plus-ja add-less-cancel-right nat-add-commute to-nat-le x-less-from-nat-a*)
qed
ultimately have *matrix-to-iarray A* !! (*to-nat x*) !! *to-nat j* = *0* **using** *least* **by** *auto*
hence *A* \$ *x* \$ *j* = *0* **unfolding** *matrix-to-iarray-nth* **.**
thus *False* **using** *Axj* **by** *contradiction*
qed
qed
hence *a* = *to-nat* (*LEAST n*. *A* \$ *n* \$ *j* ≠ *0* ∧ *i* ≤ *n*) **using** *to-nat-from-nat-id*[*OF a-less-card*] **by** *simp*
thus *?thesis* **unfolding** *least-n-def* **unfolding** *a the.simps* **.**
qed

lemma *matrix-to-iarray-Gauss-Jordan-in-ij*[*code-unfold*]:
fixes *A*::′*a*::{*field*} ^′*columns*::{*mod-type*} ^′*rows*::{*mod-type*}
assumes *not-all-zero*: ¬ (*all-zero-from-k* (*to-nat i*, *matrix-to-iarray A*) (*to-nat j*))
shows *matrix-to-iarray* (*Gauss-Jordan-in-ij A i j*) = *Gauss-Jordan-in-ij-iarrays* (*matrix-to-iarray A*) (*to-nat i*) (*to-nat j*)
proof (*unfold Gauss-Jordan-in-ij-def Gauss-Jordan-in-ij-iarrays-def Let-def*, *rule matrix-to-iarray-eq-of-fun*, *auto*)
show *vec-to-iarray* (*mult-row* (*interchange-rows A i* (*LEAST n*. *A* \$ *n* \$ *j* ≠ *0* ∧ *i* ≤ *n*)) *i* (*1* / *A* \$ (*LEAST n*. *A* \$ *n* \$ *j* ≠ *0* ∧ *i* ≤ *n*) \$ *j*) \$ *i*) =
mult-row-iarray (*interchange-rows-iarray* (*matrix-to-iarray A*) (*to-nat i*) (*least-n* (*matrix-to-iarray A*) (*to-nat i*) (*to-nat j*))) (*to-nat i*)
(*1* / *interchange-rows-iarray* (*matrix-to-iarray A*) (*to-nat i*) (*least-n* (*matrix-to-iarray A*) (*to-nat i*) (*to-nat j*)) !! *to-nat i* !! *to-nat j*) !! *to-nat i*
unfolding *matrix-to-iarray-least-n*[*OF not-all-zero*]
unfolding *matrix-to-iarray-interchange-rows*[*symmetric*]
unfolding *matrix-to-iarray-mult-row*[*symmetric*]
unfolding *matrix-to-iarray-nth*
unfolding *interchange-rows-i*
unfolding *vec-matrix* **..**
next
fix *ia*
show *vec-to-iarray*
(*row-add* (*mult-row* (*interchange-rows A i* (*LEAST n*. *A* \$ *n* \$ *j* ≠ *0* ∧ *i* ≤

n)) *i* (*1* / *A* $ (*LEAST n*. *A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*) $ *j*)) *ia i*
(− *interchange-rows A i* (*LEAST n*. *A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*) $ *ia* $ *j*) $ *ia*) =
row-add-iarray (*mult-row-iarray* (*interchange-rows-iarray* (*matrix-to-iarray A*) (*to-nat i*) (*least-n* (*matrix-to-iarray A*) (*to-nat i*) (*to-nat j*)))
(*to-nat i*) (*1* / *interchange-rows-iarray* (*matrix-to-iarray A*) (*to-nat i*) (*least-n* (*matrix-to-iarray A*) (*to-nat i*) (*to-nat j*)) !!
to-nat i !! *to-nat j*)) (*to-nat ia*) (*to-nat i*)
(− *interchange-rows-iarray* (*matrix-to-iarray A*) (*to-nat i*) (*least-n* (*matrix-to-iarray A*) (*to-nat i*) (*to-nat j*)) !!
to-nat ia !! *to-nat j*) !! *to-nat ia*
unfolding *matrix-to-iarray-least-n*[*OF not-all-zero*]
unfolding *matrix-to-iarray-interchange-rows*[*symmetric*]
unfolding *matrix-to-iarray-mult-row*[*symmetric*]
unfolding *matrix-to-iarray-nth*
unfolding *interchange-rows-i*
unfolding *matrix-to-iarray-row-add*[*symmetric*]
unfolding *vec-matrix* **..**
next
show *nrows-iarray* (*matrix-to-iarray A*) =
IArray.length (*matrix-to-iarray*
(χ *s*. *if s* = *i then mult-row* (*interchange-rows A i* (*LEAST n*. *A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*)) *i* (*1* / *interchange-rows A i* (*LEAST n*. *A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*) $ *i* $ *j*) $ *s*
else row-add (*mult-row* (*interchange-rows A i* (*LEAST n*. *A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*)) *i* (*1* / *interchange-rows A i* (*LEAST n*. *A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*) $ *i* $ *j*)) *s i*
(− *interchange-rows A i* (*LEAST n*. *A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*) $ *s* $ *j*) $ *s*))
unfolding *length-eq-card-rows nrows-eq-card-rows* **..**
qed

lemma *matrix-to-iarray-Gauss-Jordan-column-k-1*:
fixes *A*::′*a*::{*field*} ^′*columns*::{*mod-type*} ^′*rows*::{*mod-type*}
assumes *k*: *k*<*ncols A*
and *i*: *i*≤*nrows A*
shows (*fst* (*Gauss-Jordan-column-k* (*i*, *A*) *k*)) = *fst* (*Gauss-Jordan-column-k-iarrays* (*i*, *matrix-to-iarray A*) *k*)
proof (*cases i*<*nrows A*)
case *True*
show *?thesis*
unfolding *Gauss-Jordan-column-k-def Let-def Gauss-Jordan-column-k-iarrays-def*

unfolding *all-zero-from-k-eq fst-conv*
unfolding *to-nat-from-nat-id*[*OF True*[*unfolded nrows-def*]] *to-nat-from-nat-id*[*OF k*[*unfolded ncols-def*]]
unfolding *matrix-to-iarray-nrows snd-conv* **by** *simp*
next
case *False*
have *all-zero-from-k* (*nrows A*, *matrix-to-iarray A*) *k* **unfolding** *all-zero-from-k-def*

Let-def
by (*simp add*: *is-none-code(1) nrows-def nrows-iarray-def length-eq-card-rows*)
thus *?thesis*
using *i False*
unfolding *Gauss-Jordan-column-k-iarrays-def Gauss-Jordan-column-k-def Let-def*
by *auto*
qed

lemma *matrix-to-iarray-Gauss-Jordan-column-k-2*:
fixes *A*::*'a*::{*field*} ^*'columns*::{*mod-type*} ^*'rows*::{*mod-type*}
assumes *k*: *k<ncols A*
and *i*: *i≤nrows A*
shows *matrix-to-iarray* (*snd* (*Gauss-Jordan-column-k* (*i*, *A*) *k*)) = *snd* (*Gauss-Jordan-column-k-iarrays* (*i*, *matrix-to-iarray A*) *k*)
proof (*cases i<nrows A*)
case *True* **show** *?thesis*
unfolding *Gauss-Jordan-column-k-def Gauss-Jordan-column-k-iarrays-def snd-conv fst-conv Let-def*
unfolding *all-zero-from-k-eq* **unfolding** *Suc-eq-plus1*
unfolding *matrix-to-iarray-nrows*
using *matrix-to-iarray-Gauss-Jordan-in-ij*[*of from-nat i A from-nat k*]
unfolding *to-nat-from-nat-id*[*OF True*[*unfolded nrows-def*]]
unfolding *to-nat-from-nat-id*[*OF k*[*unfolded ncols-def*]] **by** *simp*
next
case *False* **show** *?thesis*
using *assms False* **unfolding** *Gauss-Jordan-column-k-def Let-def Gauss-Jordan-column-k-iarrays-def*
by (*auto simp add*: *matrix-to-iarray-nrows*)
qed

Due to the assumptions presented in $[\![?k < ncols\ ?A;\ ?i \leq nrows\ ?A]\!] \Longrightarrow matrix\text{-}to\text{-}iarray\ (snd\ (Gauss\text{-}Jordan\text{-}column\text{-}k\ (?i,\ ?A)\ ?k)) = snd\ (Gauss\text{-}Jordan\text{-}column\text{-}k\text{-}iarrays\ (?i,\ matrix\text{-}to\text{-}iarray\ ?A)\ ?k)$, the following lemma must have three shows. The proof style is similar to $?k < ncols\ ?A \Longrightarrow reduced\text{-}row\text{-}echelon\text{-}form\text{-}upt\text{-}k\ (Gauss\text{-}Jordan\text{-}upt\text{-}k\ ?A\ ?k)\ (Suc\ ?k)$
$?k < ncols\ ?A \Longrightarrow foldl\ Gauss\text{-}Jordan\text{-}column\text{-}k\ (0,\ ?A)\ [0..<Suc\ ?k] = (if\ \forall m.\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ m\ (Suc\ ?k)\ (snd\ (foldl\ Gauss\text{-}Jordan\text{-}column\text{-}k\ (0,\ ?A)\ [0..<Suc\ ?k]))\ then\ 0\ else\ mod\text{-}type\text{-}class.to\text{-}nat\ (GREATEST'\ n.\ \neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ n\ (Suc\ ?k)\ (snd\ (foldl\ Gauss\text{-}Jordan\text{-}column\text{-}k\ (0,\ ?A)\ [0..<Suc\ ?k]))) + 1,\ snd\ (foldl\ Gauss\text{-}Jordan\text{-}column\text{-}k\ (0,\ ?A)\ [0..<Suc\ ?k]))$.

lemma
fixes *A*::*'a*::{*field*} ^*'columns*::{*mod-type*} ^*'rows*::{*mod-type*}
assumes *k*: *k<ncols A*
shows *matrix-to-iarray-Gauss-Jordan-upt-k*[*code-unfold*]: *matrix-to-iarray* (*Gauss-Jordan-upt-k A k*) = *Gauss-Jordan-upt-k-iarrays* (*matrix-to-iarray A*) *k*
and *fst-foldl-Gauss-Jordan-column-k-eq*: *fst* (*foldl Gauss-Jordan-column-k-iarrays* (*0*, *matrix-to-iarray A*) [*0..<Suc k*]) = *fst* (*foldl Gauss-Jordan-column-k* (*0*, *A*)

[*0..<Suc k*])
and *fst-foldl-Gauss-Jordan-column-k-less*: *fst* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0..<Suc k*]) ≤ *nrows A*
using *assms*
proof (*induct k*)
show *matrix-to-iarray* (*Gauss-Jordan-upt-k A 0*) = *Gauss-Jordan-upt-k-iarrays* (*matrix-to-iarray A*) *0*
unfolding *Gauss-Jordan-upt-k-def Gauss-Jordan-upt-k-iarrays-def* **by** (*auto*, *metis k le0 less-nat-zero-code matrix-to-iarray-Gauss-Jordan-column-k-2 neq0-conv*)

show *fst* (*foldl Gauss-Jordan-column-k-iarrays* (*0*, *matrix-to-iarray A*) [*0..<Suc 0*]) = *fst* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0..<Suc 0*])
unfolding *Gauss-Jordan-upt-k-def Gauss-Jordan-upt-k-iarrays-def* **by** (*auto*, *metis gr-implies-not0 k le0 matrix-to-iarray-Gauss-Jordan-column-k-1 neq0-conv*)
show *fst* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0..<Suc 0*]) ≤ *nrows A* **unfolding** *Gauss-Jordan-upt-k-def* **by** (*simp add*: *Gauss-Jordan-column-k-def size1 nrows-def*)
next
fix *k*
assume (*k* < *ncols A* ⟹ *matrix-to-iarray* (*Gauss-Jordan-upt-k A k*) = *Gauss-Jordan-upt-k-iarrays* (*matrix-to-iarray A*) *k*) **and**
(*k* < *ncols A* ⟹ *fst* (*foldl Gauss-Jordan-column-k-iarrays* (*0*, *matrix-to-iarray A*) [*0..<Suc k*]) = *fst* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0..<Suc k*]))
and (*k* < *ncols A* ⟹ *fst* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0..<Suc k*]) ≤ *nrows A*)
and *Suc-k-less-card*: *Suc k* < *ncols A*
hence *hyp1*: *matrix-to-iarray* (*Gauss-Jordan-upt-k A k*) = *Gauss-Jordan-upt-k-iarrays* (*matrix-to-iarray A*) *k*
and *hyp2*: *fst* (*foldl Gauss-Jordan-column-k-iarrays* (*0*, *matrix-to-iarray A*) [*0..<Suc k*]) = *fst* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0..<Suc k*])
and *hyp3*: *fst* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0..<Suc k*]) ≤ *nrows A*
by *auto*
hence *hyp1-unfolded*: *matrix-to-iarray* (*snd* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0..<Suc k*])) = *snd* (*foldl Gauss-Jordan-column-k-iarrays* (*0*, *matrix-to-iarray A*) [*0..<Suc k*])
using *hyp1* **unfolding** *Gauss-Jordan-upt-k-def Gauss-Jordan-upt-k-iarrays-def* **by** *simp*
have *upt-rw*: [*0..<Suc* (*Suc k*)] = [*0..<Suc k*] @ [(*Suc k*)] **by** *auto*
have *fold-rw*: (*foldl Gauss-Jordan-column-k-iarrays* (*0*, *matrix-to-iarray A*) [*0..<Suc k*])
= (*fst* (*foldl Gauss-Jordan-column-k-iarrays* (*0*, *matrix-to-iarray A*) [*0..<Suc k*]), *snd* (*foldl Gauss-Jordan-column-k-iarrays* (*0*, *matrix-to-iarray A*) [*0..<Suc k*]))
by *simp*
have *fold-rw'*: (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0..<*(*Suc k*)])
= (*fst* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0..<*(*Suc k*)]), *snd* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0..<*(*Suc k*)])) **by** *simp*
show *fst* (*foldl Gauss-Jordan-column-k-iarrays* (*0*, *matrix-to-iarray A*) [*0..<Suc* (*Suc k*)]) = *fst* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0..<Suc* (*Suc k*)])

unfolding *upt-rw foldl-append* **unfolding** *List.foldl.simps* **apply** (*subst fold-rw*)
apply (*subst fold-rw'*) **unfolding** *hyp2* **unfolding** *hyp1-unfolded*[*symmetric*]
proof (*rule matrix-to-iarray-Gauss-Jordan-column-k-1*[*symmetric*, *of Suc k* (*snd* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc k*]))])
show *Suc k* < *ncols* (*snd* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc k*]))
using *Suc-k-less-card* **unfolding** *ncols-def* **.**
show *fst* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc k*]) ≤ *nrows* (*snd* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc k*])) **using** *hyp3* **unfolding** *nrows-def* **.**
qed
show *matrix-to-iarray* (*Gauss-Jordan-upt-k A* (*Suc k*)) = *Gauss-Jordan-upt-k-iarrays* (*matrix-to-iarray A*) (*Suc k*)
unfolding *Gauss-Jordan-upt-k-def Gauss-Jordan-upt-k-iarrays-def upt-rw foldl-append List.foldl.simps*
apply (*subst fold-rw*) **apply** (*subst fold-rw'*) **unfolding** *hyp2 hyp1-unfolded*[*symmetric*]
proof (*rule matrix-to-iarray-Gauss-Jordan-column-k-2*, *unfold ncols-def nrows-def*)
show *Suc k* < *CARD*(*'columns*) **using** *Suc-k-less-card* **unfolding** *ncols-def* **.**
show *fst* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc k*]) ≤ *CARD*(*'rows*)
using *hyp3* **unfolding** *nrows-def* **.**
qed
show *fst* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc* (*Suc k*)]) ≤ *nrows A*
unfolding *upt-rw foldl-append* **unfolding** *List.foldl.simps* **apply** (*subst fold-rw'*)
unfolding *Gauss-Jordan-column-k-def Let-def*
using *hyp3 le-antisym not-less-eq-eq* **unfolding** *nrows-def* **by** *fastforce*
qed

lemma *matrix-to-iarray-Gauss-Jordan*[*code-unfold*]:
fixes *A*::*'a*::{*field*} ^*'columns*::{*mod-type*} ^*'rows*::{*mod-type*}
shows *matrix-to-iarray* (*Gauss-Jordan A*) = *Gauss-Jordan-iarrays* (*matrix-to-iarray A*)
unfolding *Gauss-Jordan-iarrays-def ncols-iarray-def* **unfolding** *length-eq-card-columns*
by (*auto simp add*: *Gauss-Jordan-def matrix-to-iarray-Gauss-Jordan-upt-k ncols-def*)

14.5 Proving the equivalence between *rank* and *rank-iarray*.

First of all, some code equations are removed to allow the execution of Gauss-Jordan algorithm using iarrays

lemmas *card'-code*(*2*)[*code del*]
lemmas *rank-Gauss-Jordan-code*[*code del*]

lemma *rank-eq-card-iarrays*:
fixes *A*::*real*^*'columns*::{*mod-type*} ^*'rows*::{*mod-type*}
shows *rank A* = *card* {*vec-to-iarray* (*row i* (*Gauss-Jordan A*)) |*i*. ¬ *is-zero-iarray* (*vec-to-iarray* (*row i* (*Gauss-Jordan A*)))}
proof (*unfold rank-Gauss-Jordan-code*, *rule bij-betw-same-card*[*of vec-to-iarray*], *auto simp add*: *bij-betw-def*)
show *inj-on vec-to-iarray* {*row i* (*Gauss-Jordan A*) |*i*. *row i* (*Gauss-Jordan A*) ≠ *0*} **using** *inj-vec-to-iarray* **unfolding** *inj-on-def* **by** *blast*

fix *i* **assume** *r*: *row i* (*Gauss-Jordan A*) ≠ *0*
show ∃ *ia*. *vec-to-iarray* (*row i* (*Gauss-Jordan A*)) = *vec-to-iarray* (*row ia* (*Gauss-Jordan A*)) ∧ ¬ *is-zero-iarray* (*vec-to-iarray* (*row ia* (*Gauss-Jordan A*)))
proof (*rule exI*[*of - i*], *simp*)
show ¬ *is-zero-iarray* (*vec-to-iarray* (*row i* (*Gauss-Jordan A*))) **using** *r* **unfolding** *is-zero-iarray-eq-iff* **.**
qed
next
fix *i*
assume *not-zero-iarray*: ¬ *is-zero-iarray* (*vec-to-iarray* (*row i* (*Gauss-Jordan A*)))
show *vec-to-iarray* (*row i* (*Gauss-Jordan A*)) ∈ *vec-to-iarray* ‘ {*row i* (*Gauss-Jordan A*) |*i*. *row i* (*Gauss-Jordan A*) ≠ *0*}
by (*rule imageI*, *auto simp add*: *not-zero-iarray is-zero-iarray-eq-iff*)
qed

lemma *rank-eq-card-iarrays′*:
fixes *A*::*real*^′*columns*::{*mod-type*}^′*rows*::{*mod-type*}
shows *rank A* = (*let A′* = (*Gauss-Jordan-iarrays* (*matrix-to-iarray A*)) *in card* {*row-iarray* (*to-nat i*) *A′* |*i*::′*rows*. ¬ *is-zero-iarray* (*A′* !! (*to-nat i*))})
unfolding *Let-def* **unfolding** *rank-eq-card-iarrays vec-to-iarray-row′ matrix-to-iarray-Gauss-Jordan row-iarray-def* **..**

lemma *rank-eq-card-iarrays-code*:
fixes *A*::*real*^′*columns*::{*mod-type*}^′*rows*::{*mod-type*}
shows *rank A* = (*let A′* = (*Gauss-Jordan-iarrays* (*matrix-to-iarray A*)) *in card* {*i*::′*rows*. ¬ *is-zero-iarray* (*A′* !! (*to-nat i*))})
proof (*unfold rank-eq-card-iarrays′ Let-def*, *rule bij-betw-same-card*[*symmetric*, *of* λ*i*. *row-iarray* (*to-nat i*) (*Gauss-Jordan-iarrays* (*matrix-to-iarray A*))],
unfold bij-betw-def inj-on-def, *auto*)
fix *x y*::′*rows*
assume *x*: ¬ *is-zero-iarray* (*Gauss-Jordan-iarrays* (*matrix-to-iarray A*) !! *to-nat x*)
and *y*: ¬ *is-zero-iarray* (*Gauss-Jordan-iarrays* (*matrix-to-iarray A*) !! *to-nat y*)
and *eq*: *row-iarray* (*to-nat x*) (*Gauss-Jordan-iarrays* (*matrix-to-iarray A*)) = *row-iarray* (*to-nat y*) (*Gauss-Jordan-iarrays* (*matrix-to-iarray A*))
have *eq′*: (*Gauss-Jordan A*) $ *x* = (*Gauss-Jordan A*) $ *y* **by** (*metis eq matrix-to-iarray-Gauss-Jordan row-iarray-def vec-matrix vec-to-iarray-morph*)
hence *not-zero-x*: ¬ *is-zero-row x* (*Gauss-Jordan A*) **and** *not-zero-y*: ¬ *is-zero-row y* (*Gauss-Jordan A*)
by (*metis is-zero-iarray-eq-iff is-zero-row-def′ matrix-to-iarray-Gauss-Jordan vec-eq-iff vec-matrix x zero-index*)+
hence *x-in*: *row x* (*Gauss-Jordan A*) ∈ {*row i* (*Gauss-Jordan A*) |*i*::′*rows*. *row i* (*Gauss-Jordan A*) ≠ *0*}
and *y-in*: *row y* (*Gauss-Jordan A*) ∈ {*row i* (*Gauss-Jordan A*) |*i*::′*rows*. *row i* (*Gauss-Jordan A*) ≠ *0*}
by (*metis* (*lifting*, *mono-tags*) *is-zero-iarray-eq-iff matrix-to-iarray-Gauss-Jordan mem-Collect-eq vec-to-iarray-row′ x y*)+

show $x = y$ **using** *inj-index-independent-rows*[*OF - x-in eq'*] *rref-Gauss-Jordan*
by *fast*
qed

lemma *rank-iarrays-code*[*code-unfold*]:
rank-iarray A = *length* (*filter* (λx. $\neg$ *is-zero-iarray x*) (*IArray.list-of* (*Gauss-Jordan-iarrays A*)))
proof −
obtain *xs* **where** *A-eq-xs*: (*Gauss-Jordan-iarrays A*) = *IArray xs* **using** *list-of.cases*
by *blast*
have *rank-iarray A* = *card* {*i*. *i*<(*IArray.length* (*Gauss-Jordan-iarrays A*)) $\wedge$ $\neg$ *is-zero-iarray* ((*Gauss-Jordan-iarrays A*) !! *i*)} **unfolding** *rank-iarray-def Let-def*
..
also have ... = *length* (*filter* (λx. $\neg$ *is-zero-iarray x*) (*IArray.list-of* (*Gauss-Jordan-iarrays A*)))
unfolding *A-eq-xs* **using** *length-filter-conv-card*[*symmetric*] **by** *force*
finally show *?thesis* **.**
qed

lemma *matrix-to-iarray-rank*[*code-unfold*]:
shows *rank A* = *rank-iarray* (*matrix-to-iarray A*)
unfolding *rank-eq-card-iarrays-code rank-iarray-def Let-def*
apply (*rule bij-betw-same-card*[*of to-nat*])
unfolding *bij-betw-def* **apply** *auto* **apply** (*metis inj-onI to-nat-eq*)
unfolding *matrix-to-iarray-Gauss-Jordan*[*symmetric*] *length-eq-card-rows*
using *bij-to-nat*[**where** *?'a='b*] **unfolding** *bij-betw-def* **by** *auto*

14.6 Definitions of null space, row space and column space for matrices represented as nested iarrays.

We can't write: *dim* (*null-space-iarray* (*matrix-to-iarray A*)), because the operator *dim* can be only applied to sets whose elements belong to a type which is an instance of real vector

lemma *dim-null-space-iarray*[*code-unfold*]:
fixes *A*::*real^'columns*::{*mod-type*}*^'rows*::{*mod-type*}
shows *dim* (*null-space A*) = *ncols-iarray* (*matrix-to-iarray A*) − *rank-iarray* (*matrix-to-iarray A*)
unfolding *dim-null-space ncols-eq-card-columns matrix-to-iarray-rank* **by** *simp*

lemma *dim-col-space-iarray*[*code-unfold*]:
fixes *A*::*real^'columns*::{*mod-type*}*^'rows*::{*mod-type*}
shows *dim* (*col-space A*) = *rank-iarray* (*matrix-to-iarray A*)
unfolding *rank-eq-dim-col-space*[*of A*, *symmetric*] *matrix-to-iarray-rank* **..**

lemma *dim-row-space-iarray*[*code-unfold*]:
fixes *A*::*real^'columns*::{*mod-type*}*^'rows*::{*mod-type*}
shows *dim* (*row-space A*) = *rank-iarray* (*matrix-to-iarray A*)
unfolding *row-rank-def*[*symmetric*] *rank-def*[*symmetric*] *matrix-to-iarray-rank*

..

end

theory *Code-Real-Approx-By-Float*
imports *Complex-Main* ~~/*src*/*HOL*/*Library*/*Code-Integer*
begin

WARNING This theory implements mathematical reals by machine reals (floats). This is inconsistent. See the proof of False at the end of the theory, where an equality on mathematical reals is (incorrectly) disproved by mapping it to machine reals.

The value command cannot display real results yet.

The only legitimate use of this theory is as a tool for code generation purposes.

code-type *real*
(*SML* *real*)
(*OCaml* *float*)

code-const *Ratreal*
(*SML* *error*/ *Bad constant*: *Ratreal*)

code-const *0* :: *real*
(*SML* *0.0*)
(*OCaml* *0.0*)
declare *zero-real-code*[*code-unfold del*]

code-const *1* :: *real*
(*SML* *1.0*)
(*OCaml* *1.0*)
declare *one-real-code*[*code-unfold del*]

code-const *HOL.equal* :: *real* ⇒ *real* ⇒ *bool*
(*SML* *Real*.== ((-), (-)))
(*OCaml* *Pervasives*.(=))

code-const *Orderings.less-eq* :: *real* ⇒ *real* ⇒ *bool*
(*SML* *Real*.<= ((-), (-)))
(*OCaml* *Pervasives*.(<=))

code-const *Orderings.less* :: *real* ⇒ *real* ⇒ *bool*
(*SML* *Real*.< ((-), (-)))
(*OCaml* *Pervasives*.(<))

code-const *op* + :: *real* ⇒ *real* ⇒ *real*
(*SML* *Real*.+ ((-), (-)))

(*OCaml Pervasives*.(+.))

code-const *op* $*$ :: *real* ⇒ *real* ⇒ *real*
(*SML* *Real*.$*$ ((-), (-)))
(*OCaml Pervasives*.($*$.))

code-const *op* − :: *real* ⇒ *real* ⇒ *real*
(*SML* *Real*.− ((-), (-)))
(*OCaml Pervasives*.(−.))

code-const *uminus* :: *real* ⇒ *real*
(*SML* *Real*.~)
(*OCaml Pervasives*.(~−.))

code-const *op* / :: *real* ⇒ *real* ⇒ *real*
(*SML* *Real*.′/ ((-), (-)))
(*OCaml Pervasives*.(′/.))

code-const *HOL.equal* :: *real* ⇒ *real* ⇒ *bool*
(*SML* *Real*.== ((-:*real*), (-)))

code-const *sqrt* :: *real* ⇒ *real*
(*SML* *Math.sqrt*)
(*OCaml Pervasives.sqrt*)
declare *sqrt-def*[*code del*]

definition *real-exp* :: *real* ⇒ *real* **where** *real-exp* = *exp*

lemma *exp-eq-real-exp*[*code-unfold*]: *exp* = *real-exp*
unfolding *real-exp-def* **..**

code-const *real-exp*
(*SML* *Math.exp*)
(*OCaml Pervasives.exp*)
declare *real-exp-def*[*code del*]
declare *exp-def*[*code del*]

hide-const (**open**) *real-exp*

code-const *ln*
(*SML* *Math.ln*)
(*OCaml Pervasives.ln*)
declare *ln-def*[*code del*]

code-const *cos*
(*SML* *Math.cos*)
(*OCaml Pervasives.cos*)
declare *cos-def*[*code del*]

code-const *sin*
(*SML* *Math.sin*)
(*OCaml* *Pervasives.sin*)
declare *sin-def*[*code del*]

code-const *pi*
(*SML* *Math.pi*)
(*OCaml* *Pervasives.pi*)
declare *pi-def*[*code del*]

code-const *arctan*
(*SML* *Math.atan*)
(*OCaml* *Pervasives.atan*)
declare *arctan-def*[*code del*]

code-const *arccos*
(*SML* *Math.scos*)
(*OCaml* *Pervasives.acos*)
declare *arccos-def*[*code del*]

code-const *arcsin*
(*SML* *Math.asin*)
(*OCaml* *Pervasives.asin*)
declare *arcsin-def*[*code del*]

definition *real-of-int* :: *int* ⇒ *real* **where**
real-of-int ≡ *of-int*

code-const *real-of-int*
(*SML* *Real.fromInt*)
(*OCaml* *Pervasives.float* (*Big′-int.int′-of′-big′-int* (-)))

lemma *of-int-eq-real-of-int*[*code-unfold*]: *of-int* = *real-of-int*
unfolding *real-of-int-def* **..**

lemma [*code-unfold del*]:
0 ≡ (*of-rat 0* :: *real*)
by *simp*

lemma [*code-unfold del*]:
1 ≡ (*of-rat 1* :: *real*)
by *simp*

lemma [*code-unfold del*]:
numeral k ≡ (*of-rat* (*numeral k*) :: *real*)
by *simp*

lemma [*code-unfold del*]:
neg-numeral k ≡ (*of-rat* (*neg-numeral k*) :: *real*)

```
  by simp

hide-const (open) real-of-int

notepad
begin
  have cos (pi/2) = 0 by (rule cos-pi-half)
  moreover have cos (pi/2) ≠ 0 by eval
  ultimately have False by blast
end

end

theory Code-Real-Approx-By-Float-Addenda
  imports $ISABELLE-HOME/src/HOL/Library/Code-Real-Approx-By-Float
begin

code-const Ratreal (SML)

definition Realfract :: int ⇒ int ⇒ real
  where Realfract p q = of-int p / of-int q

code-datatype Realfract

code-const Realfract
(SML Real.fromInt -/ '// Real.fromInt -)

lemma [code]:
  Ratreal r = split Realfract (quotient-of r)
  by (cases r) (simp add: Realfract-def quotient-of-Fract of-rat-rat)

lemma [code, code del]:
  (plus :: real ⇒ real ⇒ real) = plus
  ..

lemma [code, code del]:
  (times :: real ⇒ real ⇒ real) = times
  ..

lemma [code, code del]:
  (divide :: real ⇒ real ⇒ real) = divide
  ..

lemma [code]:
  fixes r :: real
  shows inverse r = 1 / r
  by (fact inverse-eq-divide)

lemma [code, code del]:
```

(*HOL.equal* :: *real=>real=>bool*) = (*HOL.equal* :: *real* => *real* => *bool*)
..

lemma [*code*, *code del*]:
($minus$ :: $real \Rightarrow real \Rightarrow real$) = $minus$
..

lemma [*code*, *code del*]:
($uminus$ :: $real \Rightarrow real$) = $uminus$
..

end

15 The Field of Integers mod 2

theory *Bit*
imports *Main*
begin

15.1 Bits as a datatype

typedef *bit* = *UNIV* :: *bool set* **..**

instantiation *bit* :: {*zero*, *one*}
begin

definition *zero-bit-def*:
0 = *Abs-bit False*

definition *one-bit-def*:
1 = *Abs-bit True*

instance ..

end

rep-datatype *0*::*bit* *1*::*bit*
proof −
fix *P* **and** *x* :: *bit*
assume *P* (*0*::*bit*) **and** *P* (*1*::*bit*)
then have $\forall\, b.$ *P* (*Abs-bit b*)
unfolding *zero-bit-def one-bit-def*
by (*simp add*: *all-bool-eq*)
then show *P x*
by (*induct x*) *simp*
next
show (*0*::*bit*) $\neq$ (*1*::*bit*)
unfolding *zero-bit-def one-bit-def*

by (*simp add*: *Abs-bit-inject*)
qed

lemma *bit-not-0-iff* [*iff*]: (*x*::*bit*) $\neq$ *0* $\longleftrightarrow$ *x* = *1*
by (*induct x*) *simp-all*

lemma *bit-not-1-iff* [*iff*]: (*x*::*bit*) $\neq$ *1* $\longleftrightarrow$ *x* = *0*
by (*induct x*) *simp-all*

15.2 Type *bit* forms a field

instantiation *bit* :: *field-inverse-zero*
begin

definition *plus-bit-def*:
x + *y* = *bit-case y* (*bit-case 1 0 y*) *x*

definition *times-bit-def*:
x $*$ *y* = *bit-case 0 y x*

definition *uminus-bit-def* [*simp*]:
$-$ *x* = (*x* :: *bit*)

definition *minus-bit-def* [*simp*]:
x $-$ *y* = (*x* + *y* :: *bit*)

definition *inverse-bit-def* [*simp*]:
inverse x = (*x* :: *bit*)

definition *divide-bit-def* [*simp*]:
x / *y* = (*x* $*$ *y* :: *bit*)

lemmas *field-bit-defs* =
plus-bit-def times-bit-def minus-bit-def uminus-bit-def
divide-bit-def inverse-bit-def

instance proof
qed (*unfold field-bit-defs*, *auto split*: *bit.split*)

end

lemma *bit-add-self*: *x* + *x* = (*0* :: *bit*)
unfolding *plus-bit-def* **by** (*simp split*: *bit.split*)

lemma *bit-mult-eq-1-iff* [*simp*]: *x* $*$ *y* = (*1* :: *bit*) $\longleftrightarrow$ *x* = *1* $\wedge$ *y* = *1*
unfolding *times-bit-def* **by** (*simp split*: *bit.split*)

Not sure whether the next two should be simp rules.

lemma *bit-add-eq-0-iff*: *x* + *y* = (*0* :: *bit*) $\longleftrightarrow$ *x* = *y*

unfolding *plus-bit-def* **by** (*simp split*: *bit.split*)

lemma *bit-add-eq-1-iff*: $x + y = (1 :: bit) \longleftrightarrow x \neq y$
unfolding *plus-bit-def* **by** (*simp split*: *bit.split*)

15.3 Numerals at type *bit*

All numerals reduce to either 0 or 1.

lemma *bit-minus1* [*simp*]: $-1 = (1 :: bit)$
by (*simp only*: *minus-one* [*symmetric*] *uminus-bit-def*)

lemma *bit-neg-numeral* [*simp*]: (*neg-numeral w* :: *bit*) = *numeral w*
by (*simp only*: *neg-numeral-def uminus-bit-def*)

lemma *bit-numeral-even* [*simp*]: *numeral* (*Num.Bit0 w*) = (*0* :: *bit*)
by (*simp only*: *numeral-Bit0 bit-add-self*)

lemma *bit-numeral-odd* [*simp*]: *numeral* (*Num.Bit1 w*) = (*1* :: *bit*)
by (*simp only*: *numeral-Bit1 bit-add-self add-0-left*)

end

theory *Code-Bit*
imports $*ISABELLE-HOME/src/HOL/Library/Bit*
begin

Implementation for the field of integer numbers module 2. Experimentally we have checked that this implementation is faster than using booleans or implementing the operations by tables.

code-datatype *0*::*bit* (*1*::*bit*)

code-type *bit*
(*SML IntInf.int*)
code-const *0*::*bit*
(*SML 0*)
code-const *1*::*bit*
(*SML 1*)

code-const *op* + :: *bit* => *bit* => *bit*
(*SML IntInf.rem* ((*IntInf*.+ ((-), (-))), *2*))
code-const *op* ∗ :: *bit* => *bit* => *bit*
(*SML IntInf*.∗ ((-), (-)))
code-const *op* / :: *bit* => *bit* => *bit*
(*SML IntInf*.∗ ((-), (-)))
end

theory *Examples-On-Code-Generation*
imports

```
 Gauss-Jordan-IArrays
 Code-Real-Approx-By-Float-Addenda
 Code-Bit
begin


definition iarray-of-iarray-to-list-of-list :: 'a iarray iarray => 'a list list
  where iarray-of-iarray-to-list-of-list A = map IArray.list-of (map (op !! A)
[0..<IArray.length A])

definition matrix-z2=IArray[IArray[0,1], IArray[1,1::bit]]
definition matrix-rat=IArray[IArray[0,1], IArray[1,−2::rat]]
definition matrix-real=IArray[IArray[0,1], IArray[1,−2::real]]


export-code matrix-z2
  in SML module-name matrix-z2 file matrix-z2.sml

export-code matrix-rat
  in SML module-name matrix-rat file matrix-rat.sml

export-code matrix-real
  in SML module-name matrix-real file matrix-real.sml

definition print-result-Gauss A = iarray-of-iarray-to-list-of-list (Gauss-Jordan-iarrays
A)
definition print-rank A = rank-iarray A

definition print-result-z2 = print-result-Gauss (matrix-z2)
definition print-result-rat = print-result-Gauss (matrix-rat)
definition print-result-real = print-result-Gauss (matrix-real)

definition print-rank-z2 = print-rank (matrix-z2)
definition print-rank-rat = print-rank (matrix-rat)
definition print-rank-real = print-rank (matrix-real)

export-code
  print-rank-real
  print-rank-rat
  print-rank-z2
  print-result-real
  print-result-rat
  print-result-z2
 in SML module-name GJ-GENERATED-CODE file GJ-GENERATED-CODE.sml

end
```

References

[1] S. Axler. *Linear Algebra Done Right.* Springer, 2nd edition, 1997.

[2] M. S. Gockenbach. *Finite Dimensional Linear Algebra.* CRC Press, 2010.